UNIVERSISDAD POLITÉCNICA DE MADRID

E.T.S.I. INFORMÁTICOS

Master in Software and Systems

MASTER THESIS

Design and implementation of a modular interface to integrate CLP and tabled execution

Author: Joaquín Arias Herrero

Supervisor: Manuel Carro Liñares

MADRID, JULY, 2015

Abstract

Logic programming (LP) is a family of high-level programming languages which provides high expressive power. With LP, the programmer writes the properties of the result and / or executable specifications instead of detailed computation steps.

Logic programming systems which feature tabled execution and constraint logic programming have been shown to increase the declarativeness and efficiency of Prolog, while at the same time making it possible to write very expressive programs. Tabled execution avoids infinite failure in some cases, while improving efficiency in programs which repeat computations. CLP reduces the search tree and brings the power of solving (in)equations over arbitrary domains.

Similarly to the LP case, CLP systems can also benefit from the power of tabling. Previous implementations which take ful advantage of the ideas behind tabling (e.g., forcing suspension, answer subsumption, etc. wherever it is necessary to avoid recomputation and terminate whenever possible) did not offer a simple, well-documented, easy-to-understand interface. This would be necessary to make the integratation of arbitrary CLP solvers into existing tabling systems possible. This clearly hinders a more widespread usage of the combination of both facilities.

In this thesis we examine the requirements that a constraint solver must fulfill in order to be interfaced with a tabling system. We propose and implement a framework, which we have called Mod TCLP, with a minimal set of operations (e.g., entailment checking and projection) which the constraint solver has to provide to the tabling engine.

We validate the design of Mod TCLP by a series of use cases: we re-engineer a previously existing tabled constrain domain (difference constraints) which was connected in an ad-hoc manner with the tabling engine in Ciao Prolog; we integrate Holzbauer's CLP(Q) implementation with Ciao Prolog's tabling engine; and we implement a constraint solver over (finite) lattices. We evaluate its performance with several benchmarks that implement a simple abstract interpreter whose fixpoint is reached by means of tabled execution, and whose domain operations are handled by the constraint over (finite) lattices, where TCLP avoids recomputing subsumed abstractions.

Resumen

La programación lógica con restricciones (CLP) y la tabulación son extensiones de la programación lógica que incrementan la declaratividad y eficiencia de Prolog, al mismo tiempo que hacen posible escribir programas más expresivos.

Las implementaciones anteriores que integran completamente ambas extensiones, incluyendo la suspensión de la ejecución de objetivos siempre que sea necesario, la implementación de inclusión (*subsumption*) de respuestas, etc., en todos los puntos en los que sea necesario para evitar recomputaciones y garantizar la terminación cuando sea posible, no han proporcionan una interfaz simple, bien documentada y fácil de entender. Esta interfaz es necesaria para permitir integrar resolutores de CLP arbitrarios en el sistema de tabulación. Esto claramente dificulta un uso más generalizado de la integración de ambas extensiones.

En esta tesis examinamos los requisitos que un resolutor de restricciones debe cumplir para ser integrado con un sistema de tabulación. Proponemos un esquema (y su implementación), que hemos llamado Mod TCLP, que requiere un reducido conjunto de operaciones (en particular, y entre otras, *entailment* y proyección de almacenes de restricciones) que el resolutor de restricciones debe ofrecer al sistema de tabulación.

Hemos validado el diseño de Mod TCLP con una serie de casos de uso: la refactorización de un sistema de restricciones (*difference constraints*) previamente conectado de un modo ad-hoc con la tabulación de Ciao Prolog; la integración del sistema de restricciones CLP(Q) de Holzbauer; y la implementación de un resolutor de restricciones sobre retículos finitos. Hemos evaluado su rendimiento con varios programas de prueba, incluyendo la implementación de un intérprete abstracto que alcanza su punto fijo mediante el sistema de tabulación y en el que las operaciones en el dominio son realizadas por el resolutor de restricciones sobre retículos (finitos) donde TCLP evita la recomputación de valores abstractos de las variables ya contenidos en llamadas anteriores.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Logic programming (LP) [19] is a declarative languaje based on the first order logic. A logic program consistis of a collection of Horn clauses, the theory, and a query which uses the theory to search alternative ways to be satisfy. LP gives a simple declarative and procedural semantics with high expressive power. With LP instead of detailed computational steps the programmer writes the desired properties of the result and / or the executable specifications.

Tabling [29, 32] is an execution strategy for logic programs which can suspend calls which would cause infinite recursion and saves calls and answers in order to reuse them in future calls and to resume suspended calls. Tabling has several advantages over the SLD resolution strategy (the standard resolution strategy of Prolog): it can avoid some infinite failures and improve efficiency in programs which repeat computations. It helps to make logic programs less dependent on the order of clauses and / or goals in a clause, thereby bringing operational and declarative semantics closer together.Tabled evaluation has been successfully applied in contexts which include deductive databases, program analysis, semantic Web reasoning, and model checking [9, 23, 33, 35].

Constraint Logic Programming (CLP) [17] is a natural extension of logic programming (LP) which has attracted much attention since the late 1980's. CLP languages are a class of programming languages which apply, by means of efficient, incremental constraint solving techniques (to solve (in)equations over arbitrary domains) increasing: the expressive power and declarativeness of LP, and the execution efficiency by reducing the search tree.

The interest of combining tabling and CLP stems from the fact that, similarly to the LP case, CLP systems can also benefit from the power of tabling. In other words, tabling can further increase the declarativeness and expressiveness and in many cases also their efficiency [4, 8, 26].

The theoretical basis of TCLP [5, 31] were laid out in the framework of bottom-up evaluation of Datalog systems, where soundness, completeness, and termination properties were established. While that work does not cover the full case of logic programming (due to, e.g., the restrictions on non-interpreted functions), it does show that some basic operations that the constraint do-

main needs to offer (e.g. projection and entailment checking) are necessary in order to ensure completeness w.r.t. the declarative semantics.

TCLP can bring benefits to several areas. Following [4] we will briefly cite some of them:

Constraint Databases [18] where assignments to atomic values are generalized to constraints applied to variables, which provides more compact representations and increases expressiveness [31]. Database evaluation in principle proceeds bottom-up, which ensures termination in this context. However, in order to speed up query processing and spend fewer resources, top-down evaluation is also applied, where tabling can be used to avoid loops. In this setting, TCLP is necessary to capture the semantics of the constraint database [31].

Datalog, syntactically a subset of Prolog, is often used as a query language for deductive databases and is generalized to Datalog^D to formalize reasoning about the databases. Just as Datalog is a restricted form of Logic Programming, Datalog^D is a restricted form of CLP. The restrictions enforce programs to have finite interpretations. Due to the finiteness properties, queries on Datalog^D programs can be resolved by bottom-up computation rather than the usual top-down goal-directed computation of CLP. The former has the advantage that it terminates for Datalog^D programs, whereas the latter may get stuck in infinite loops. However, a goal-directed approach usually obtains the desired result much faster and uses less space. Infinite loops in LP are worked around by tabling. TCLP generalizes Datalog^D in the same way than tabling LP generalizes Datalog.

Timed Automata [1,2] are used to represent and verify the behavior of real-time systems. Checking reachability (to verify safety properties) requires accumulating and solving constraints (with CLP) and testing for loops (with tabling). Checking the satisfiability of edge constraints w.r.t. a given state (needed by TCLP to break loops) needs constraint projection and entailment. TCLP needs constraint projection and entailment to optimize loop detection and, in some cases, to actually detect infinite loops and non-reachable states.

Abstract Interpretation which requires a fixpoint procedure is often implemented using memo tables and dependency tracking [13] which are very similar to a tabling procedure [30]: repeated calls have to be checked and accumulated information is reused. Some abstract domains [22] have a direct representation as numerical constraints; therefore, the implementation of abstract interpreters can in principle take advantage of TCLP. Finally, [31] considered TCLP as an alternative approach to implementing abstract interpretation: constraints abstract concrete values and tabling takes care of fixpoints. Section 5.3 shows preliminary results of the comparative of a tabled abstract interpreter versus a TCLP abstract interpreter.

Previous TCLP frameworks which feature a complete treatment of constraint projection and / or entailment [4] focused on how the tabling algorithm and its implementation had to be adapted to use constraints. As a result, although being generic frameworks, they were not designed to be

extensible, and adding new constraint domains was a difficult task, since the implementations of the constraint domain and tabling were intimately intertwined.

In this work we focus on the constraint solver and view it as a *server* of the tabling engine; we define a set of operations that the constraint solver has to provide in order to easily integrate with a tabling engine.

We have validated the flexibility, generality, and efficiency of our design (through a framework implemented in Ciao Prolog [14][1] that we have called Mod TCLP) by interfacing three non-trivial examples with the Ciao tabling engine to allow tabled execution of three constraint systems: a port of difference constraints solver [11] first presented in [4]; an interface to the well-known implementation of CLP(Q/R) [15,16]; and an abstract domain for signs for an abstract interpreter. We evaluate our implementation with several benchmarks.

In the next chapters, we present the contribution of this thesis. Chapter 2 presents the theoretical bases and the expressiveness of Prolog, tabling, CLP and TLCP and shows the state of the art of TCLP. Chapter 3 describes the design, implementation and flowchart of our framework Mod TCLP. In Chapters 4 and 5 we validate and evaluate the implementation by cases use and finally in Chapter 6, we conclude this thesis reflecting the results and presenting future research and practice application of this work.

[1] The most current version of Ciao Prolog is available at `http://www.ciao-lang.org`

Chapter 2

Background and Motivation

This chapter presents: (i) the terminology and basic operations performed by Prolog, tabling, the constraint solvers and TCLP and (ii) three cases study (reachability, distance-bounded reachability and left-recursion) to evaluate the benefits obtained with these extensions of Prolog.

2.1 Prolog (LP)

Prolog [27] is the most popular logic programming (LP) language, which gives a high expressive power. With LP instead of detailed computation steps the programmer writes the properties of the result and / or executable specifications.

A logic program in Pure Prolog, consists of a collection of Horn clauses. There are two types of clauses: (i) rules of the form `Head :- Body.` which means "Head is true if Body is true" and (ii) facts which are clauses with empty bodies (e.g. `edge(a,b)` which means that there is an edge from node `a` to node `b`). A rule's body consists of a conjunction of calls to predicates (named goals).

The terms in Prolog are: (i) the variables (strings beginning with an upper-case letter or underscore e.g. `X`), (ii) atom (e.g. `a`), (iii) numbers (integer or float) and (iv) functor (defined by a name (atom) and a number of arguments (terms) e.g. `padre(juan).`).

The execution is initiated by a query of the form `?- Goal(s).` (e.g. `?- edge(a, X).` is asking what node `X` has an edge from node `a`). Logically, the Prolog engine tries to find a resolution refutation of the negated query. If the negated query can be refuted by an appropriate variable bindings in place, it means that this variable binding is a logical consequence of the program (an answer to the query) and is said to have succeeded returning it to the user. Prolog uses unification to bindings the variables of a query's / predicate's call with the head of the rules. When there are multiple clause heads that match a given call, Prolog creates a choice-point and continues with the goals of the first alternative. If one goal fails, the variable bending made from the recent choice-point are undone

and the execution continues with the next alternative (this operation is called backtracking.

The standard resolution strategy of Prolog is SLD (Selecting Linear Definite), which defines the search tree (order in the choice-point) of alternative clauses using top-down evaluation. It resolves the goals of a clause from the left to right. The exploration of the search tree is therefor made depth-first, which is efficient in memory but incomplete if the search tree contains an infinite branch.

1. Reachable nodes in a directed graph Fig. 2.1 is a simple program graph problem that returns the nodes in a directed graph that are reachable at starting from another node in the graph. It is the transitive closure (reach/2) of the edge/2 relation, which represents a directed edge from a node (X) to other node (Y).

```
edge(a,b).
edge(b,a).
```

```
reach(X,Y) :-
   edge(X,Z),
   reach(Z,Y).
reach(X,Y) :-
   edge(X,Y).
```

Figure 2.1: Reach/2 transitive closure.

The query to obtain the reachable nodes from node a is ?- reach(a,X)., which does not terminate under SLD evaluation, as the graph represented by the edge/2 relation contains a cycle. Fig 2.3 (left) shows the search tree of this query where in the 5^{th} step the initial call is repeated and the evaluation enter in a loop.

Fig 2.2 shows a variant of the program if Fig 2.1 which uses (i) a list to store previous calls to avoid entering in a loop or returning repeated answers and (ii) the predicate member/2 which checks if an element is member of a list.

```
edge(a,b).
edge(b,a).


member(A,[A | _ ]).
member(A,[_ | Xs]) :-
   member(A, Xs).
```

```
reach(X, Y) :-
        reach_aux(X, Y, [X]).

reach_aux(X, Y, List) :-
        e(X, Z),
        \+ member2(Z, List),
        reach_aux(Z, Y, [Z | List]).
reach_aux(X, Y, _) :-
        e(X, Y).
```

Figure 2.2: Variant of reach/2 with an accumulative list.

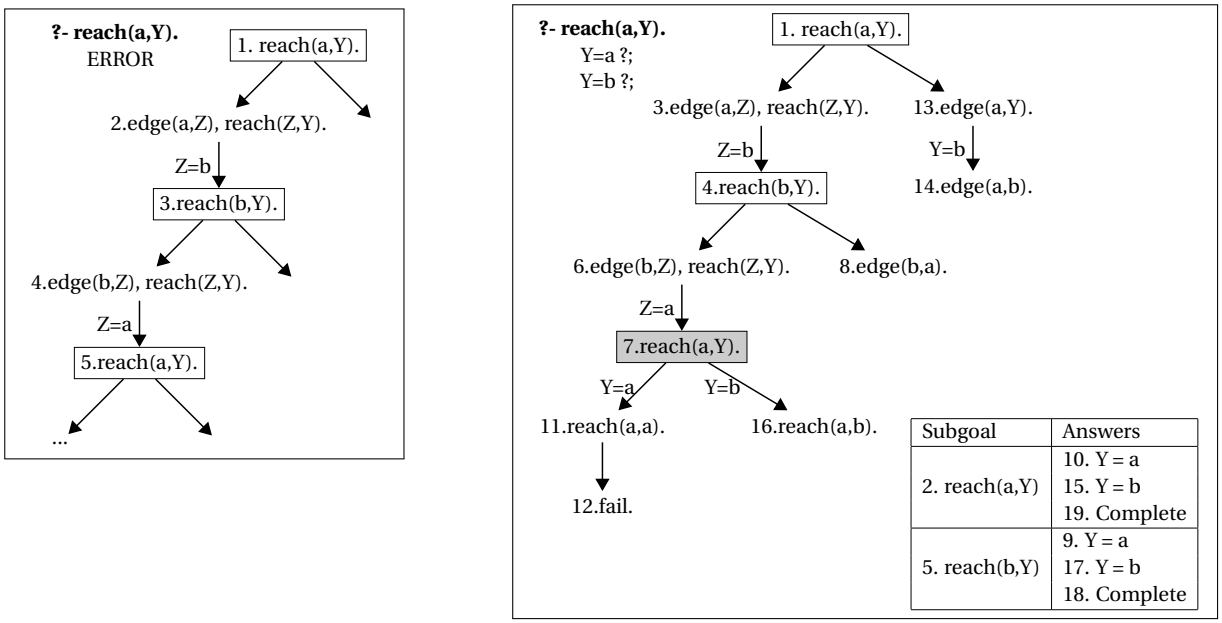

Figure 2.3: Search tree: LP (left) and Tabling (right).

2.2 Tabling

The tabled execution (tabling), as we said, is a strategy which checks for repeated call patterns and suspends the execution when a repeated call is found. Searching and storing call patterns is implemented using a global table, usually implemented as a trie [24]. Each leaf of the trie corresponds to a *generator* (the first occurrence of a tabled call), and answers to a call are also stored in the trie. The trie mechanisms avoid storing repeated answers.

To use the tabled extension in Ciao the programmer has to introduce two directives: (i) to import the tabling package and (ii) to specify which predicates shall be executed with tabling:

```
:- use_package(tabling).
:- table reach/2.
```

Adding this directives to the reach/2 program in Fig. 2.1 the query ?- reach(a,X). will be executed using the tabled execution and Fig. 2.3 (right) shows the tree search of this execution. Thanks to the tabling engine, in step number 7 this call is suspended and the search not enter a loop.

The *suspension* of a call takes place when it is a variant (a syntactically equal modulo variable renaming) of a previous generator. As we saw in step number 7 of Fig 2.3 the new call is termed a *consumer* and its execution suspends, waiting for answers to the generator call. Suspension needs to protect memory areas from backtracking by e.g. stashing them away so that they can be recovered later on when the suspended call needs to be resumed, upon the generation of new answers.

The execution continues by backtracking (step number 8) and finds the first answer ($Y = a$) for reach(b,Y) which inserted in the trie in the 9^{th} step, and for reach(a,Y) inserted in the trie in the 10^{th} step. During the insertion of new answers in the trie, repeated answers can be detected and

discarded in order to avoid repetitions.

When a generator has finitely finished exploring all of its clauses the execution of its consumers is resumed. To resume the consumer (step number 11) with the answer $\mathsf{Y} = \mathsf{a}$ stored by the generator. Then a new answer is found ($\mathsf{Y} = \mathsf{a}$) but it fails because it is already stored in the trie. By backtracking, in step number 14, a new answer ($\mathsf{Y} = \mathsf{b}$) is found for $\mathsf{reach}(\mathsf{a},\mathsf{Y})$ which is inserted in the 15^{th} step. The tabling engine resumes the consumer and inserts a new answer ($\mathsf{Y} = \mathsf{b}$) for $\mathsf{reach}(\mathsf{b},\mathsf{Y})$. There are no more clauses and the generator $\mathsf{reach}(\mathsf{b},\mathsf{Y})$ does not depend on previous generators; it is them termed complete. Then the generator $\mathsf{reach}(\mathsf{a},\mathsf{X})$ is also termed complete and it returns the answers to the query.

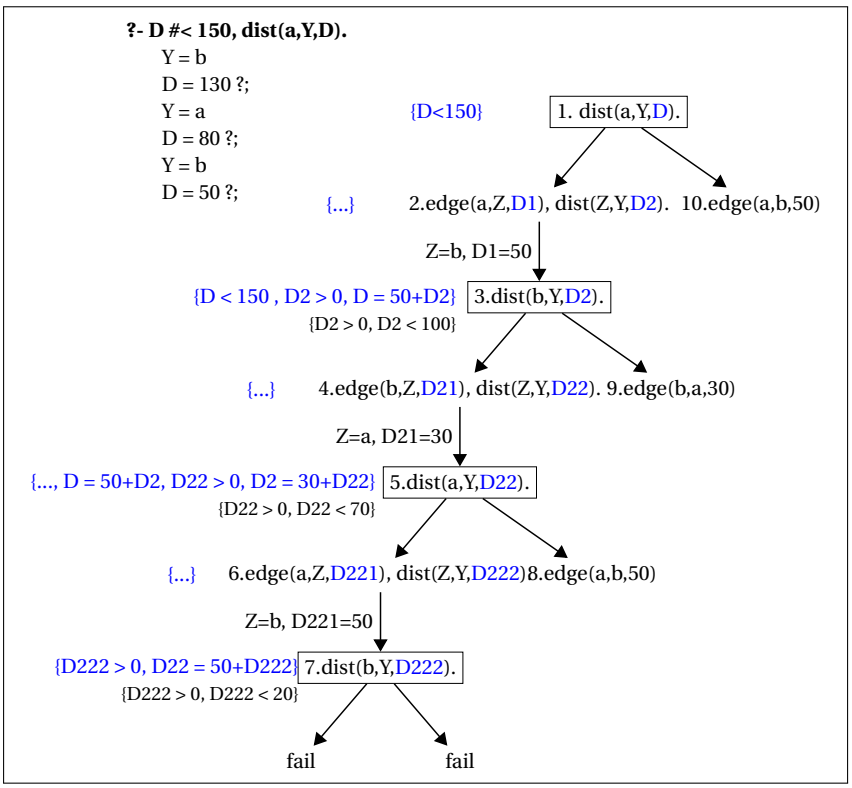

Figure 2.4: Search tree: CLP execution.

2.3 Constraint Logic Programming (CLP)

Constraint programming enhance programming languages with the ability to solve constraints ((in)equations over arbitrary domains) [10, 21].

A mathematical equation (X=Y+2) is an example of constraint . The constraint domain specifies the syntax: X,Y are variables, 2 is a constant, + is a function and = is a constraint relation. The variable's domain determines the values that a variable can take.

A constraint store is a logical conjunction of constraints. There are several relationships of interest between constraint stores S_a and S_b: They are equivalent ($S_a \leftrightarrow S_b$) if they have the same set of solutions. The constraint store S_a is entailed by S_b ($S_a \sqsubseteq S_b$) if any solution in S_a is also a solution in S_b.

Constraint solvers commonly feature some basic operations, among them satisfiability checking (determine if a set of constraints has a solution), propagation (carry changes in variable domains to other constraints), simplification (rewriting a constraint when a simpler form exists), projection (find a store which is a reduction of a previous store but reducing the variables to a given set), and optimization (finding the best solution according to some criterion).

The projection of a constraint store S_1 onto a set of variables V where $V \subseteq \mathrm{var}(S_1)$ is a new constraint store S_2 involving only the variables in V such that: (i) any solution of S_2 is also a solution of S_1 and (ii) a valuation over V which is solution in S_2 is a partial solution of S_1. Fourier's algorithm

([21] p.55-57) for variable elimination can be used to project a conjunction of lineal inequalities onto a set of variables.

Constraint propagation is the main feature of any constraint solver which tries to detect local inconsistency earlier. There are several algorithms for propagating constraints [3], e.g. indexicals used in a number of finite domain solver such as clp(fd) [6]. This propagators acts as co-routine performing incremental constraint solving and removing values from domains. It is wake up by changes in the domain of a variable(s) in the store. Then in a loop re-executed the constraints which involve these variable(s) until no changes happens to the domains of the variables in the store. This procedure also means that the textual order of the constraint does not impact the final solution.

2. Distance-bounded reachable nodes in a weighted directed graph. Fig 2.5 is a distance-bounded graph program (on the left in LP and on the right in CLP(Q)) that returns the nodes in a directed weighted graph that are reachable at starting from another node in the graph, and the distance between them. In order to have a finite answer space the distance is bounded with a limit. The edge/3 relation, represent the directed edge and the length of the edge.

```
edge(a,b,50).
edge(b,a,30).

dist(X,Y,Dmax,D) :-
    Dmax > 0,
    edge(X,Z,D1),
    NewDmax is Dmax - D1,
    dist(Z,Y,NewDmax,D2),
    D is D1 + D2.
dist(X,Y,Dmax,D) :-
    edge(X,Y,D),
    Dmax > D.
```

```
edge(a,b,50).
edge(b,a,30).

dist(X,Y,D) :-
    D1 #> 0,
    D2 #> 0,
    D #= D1 + D2,
    edge(X,Z,D1),
    dist(Z,Y,D2).
dist(X,Y,D) :-
    edge(X,Y,D).
```

Figure 2.5: Dist/4 in LP (left), dist/3 in CLP (right).

To obtain the reachable nodes from node a and the distance between them with a limit of 150, the main differences are: (i) the predicate in LP needs one extra argument to pass the distance limit and (ii) the query in LP is ?- dist(a,Y,150,D) and in CLP is ?- D #< 150, dist(a,Y,D).

CLP gives more expressiveness because queries such as ?- D #> 60, D #< 150, dist(a,Y,D) return the nodes and the distance in the interval $60 < D < 150$ while in LP it is necessary to: (i) change the program adding a new argument (dist/5) or (ii) make the previous query and apply a filter (?- dist(a,Y,D,150), $D < 60$), which is less efficient.

Another advantage of CLP versus LP in this example is that the search tree is smaller. In CLP the restrictions can be define and checked before the query / user predicates in a natural way. In LP, as we saw, the program becomes more complex. When the constraint solver adds a constraint, if checks if it is consistent with the current constraint store; if not it fails and the execution continues by backtracking with the next clause.

Fig. 2.4 shows the search tree of the query $\mathsf{D} \ \#< \ 150, \ \mathsf{dist}(\mathsf{a},\mathsf{Y},\mathsf{D}).$ under CLP(Q). In the CLP's search tree, blue is used to identify: (i) restricted variables of the calls and (ii) the representation of the constraint store at the moment of the call (next to each call). To make it easier to reader the comparison between the constraint involve in the call, the constraint store in black represent the projection onto the variable of the call.

Initially the query has the constraint $\{\mathsf{D} < 150\}$ and in the 5^{th} step the call $\mathsf{dist}(\mathsf{a},\mathsf{Y},\mathsf{D2})$ has a more concrete restriction over D2, $\{\mathsf{D2} > 0, \ \mathsf{D2} < 70\}$. In the 7^{th} step the call fails because there are no edge that unify with a distance between 0 and 20. Then on backtracking it chooses the base clause (step number 8) and finds the first answer ($\mathsf{Y} = \mathsf{b}, \ \mathsf{D} = 130$). Steps 9 and 10 give the other answers.

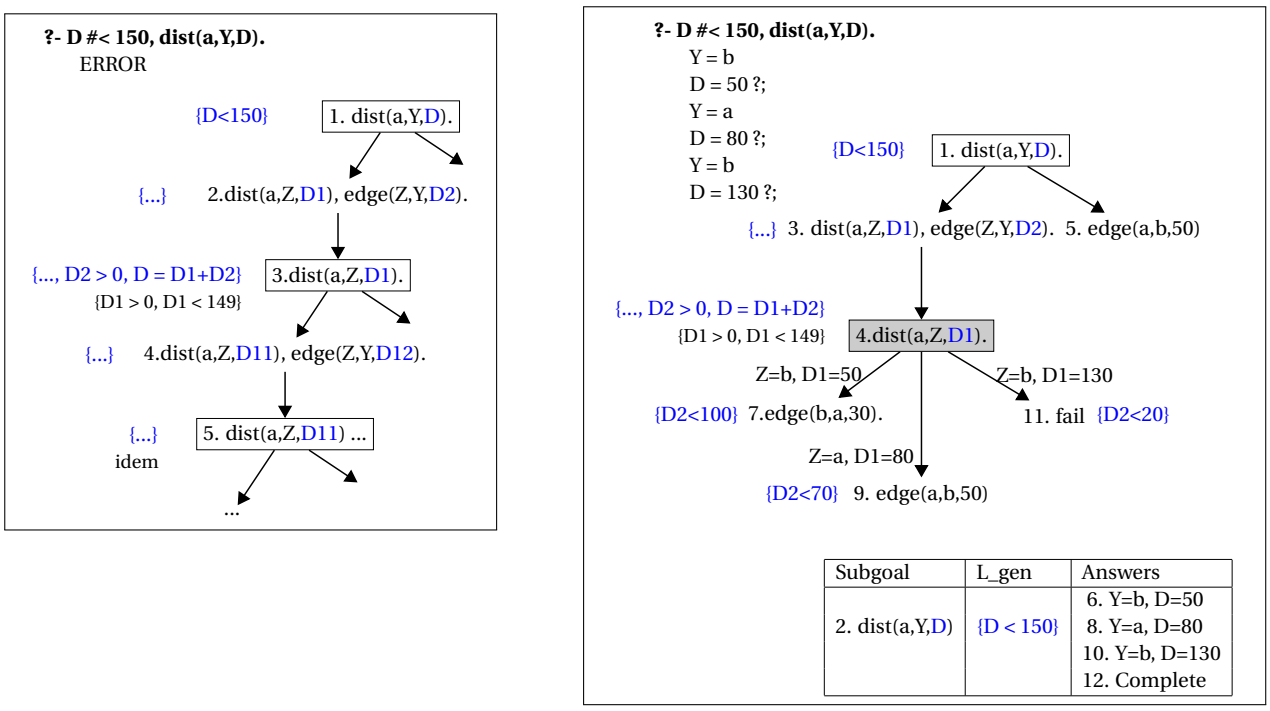

Figure 2.6: Search tree: CLP (left) and TCLP (right).

2.4 Tabling and Constraints (TCLP)

The integration of tabling and constraints (TCLP) allows the suspension of calls (consumers) when the constraint store of the call is entailed by the constraint store of a previous call (generator). To retain the completeness properties of tabling when using constraints, repeated call checker is executed in two steps (i) tabling's variant check (modulo free and restricted variables renaming) and (ii) constraint entailment to check the constraint stores of the call/answer to suspend/discard the more particulars.

3. Left-recursion In the previous examples the programs were written with right recursion. It is know that left recursion under SLD resolution often generates infinite branches and Prolog enters a loop. There exist grammars and algorithms that required left-recursive programs to , so the programmer should apply algorithm to remove the direct and / or the indirect left recursion, resulting less declarative and readable programs.

Fig. 2.7 shows the dist/3 predicates with the user predicates written with left-recursion. The query (?- D #< 150, dist(a,Y,D).) in CLP (see Fig. 2.6 (left)) enters a loop, because the constraint store does not change in each recursive call.

Fig. 2.6 (right) shows the search tree of the query ?- D #< 150, dist(a,Y,D). under TCLP(Q) and illustrate how TCLP is able to compute using left-recursive programs. In the 1^{st} step the tabling engine saves the generator call and the projected constraint store ($\{D < 150\}$). When in the 4^{th} step the tabling engine suspends the execution because: (i) the pattern of the call matches the

```
edge(a,b,50).
edge(b,a,30).

dist(X,Y,D) :-
    D1 #> 0,
    D2 #> 0,
    D #= D1 + D2,
    dist(X,Z,D1),
    edge(Z,Y,D2).
dist(X,Y,D) :-
    edge(X,Y,D).
```

Figure 2.7: Dist/3 (left recursion).

generator and (ii) the constraint store ({..., D1 $>$ 0, D1<150}) is entailed by the projected store of the generator. Then in step number 5, the tabling engine finds the first answer and after resuming the consumer, it finds the other two answers. In the 11^{th} step when TCLP applies the third answer, it fails because there are no clause that unifies with edge(a,b,D) {D $<$ 20}. There are no more clauses and the generator does not depend on previous generators, so it is termed complete and returns the answers to the query.

The combination of tabling and CLP not only gives all the solutions, but also the final search tree is smaller that the search tree in Fig. 2.4 corresponding to the right-recursive version of the dist/3 executed with CLP.

2.5 TCLP's State of the Art

The previous sections illustrated the declarativeness and expressiveness of TCLP and the improvement in termination properties versus the other operational semantics. Table 2.1 summarizes these results: (i) the reachable nodes program reach/2 needs an extra argument to store the visited nodes and the predicatemember/2 to terminate in SLD, (ii) the distance-bounded program needs an extra argument to reduce the answer space and terminate, and (iii) only TCLP is able to execute program with left recursion. So TCLP does not need to reformulate the natural ways solving these problems, resulting in more declarative and readable programs.

	Prolog	Tabling	CLP	TCLP
1. Reachable nodes	x	✓	x	✓
2. Distance-bounded	x	x	✓	✓
3. Left recursion	x	x	x	✓

Table 2.1: Comparative termination properties.
The ✓means termination in more cases.

The initial ideas to combine tabling and constraints originate in [18], where a variant of Datalog featuring constraints was proposed. The time and space problems associated with the bottom-up evaluation of Datalog were worked around in [31] where a top-down evaluation strategy featuring tabling was proposed.

XSB [28] was the first logic programming system which aimed at providing tabled CLP as a generic feature, instead of resorting to ad-hoc adaptations. This was done by extending XSB with attributed variables [8], one of the most popular mechanism to implement constraint solvers in Prolog. Native support for attributed variables brings advantages at the implementation level. However, one of its drawbacks is that it only uses variant call checking (including those with constraints), instead of entailment checking of calls / answers. This makes programs terminate in less cases and take longer in other cases.

A general framework for CHR under tabled evaluation is described in [26]. This approach brings the flexibility that CHR provides for writing constraint solvers, but it also lacks call entailment checking and enforces total call abstraction: all constraints are removed from calls before executing them, which can result in non-termination w.r.t. systems which do not use call abstraction. Besides, the need to change the representation between CHR and Herbrand terms takes a toll in performance.

Last, TCLP [4] features entailment checking both for calls and for answers, executes calls with all of constraints, and exhibits good performance. However, it does not clarify which operations must be present in the constraint solver, and is not focused on a modular design.

Chapter 3

The Generic Interface Mod TCLP

In this chapter we present Mod TCLP which focuses on the constraint solver and views it as a *server* of the tabling engine. We design this framework to be extensible and to allow the addition of new constraint domains in a easy way. In section 3.1 we describe the design of Mod TCLP (its interface predicates and arguments) and detail the implementation of the main features (projection and entailment). Section 3.2 presents the program transformation applied to the original programs to execute them under tabling. Section 3.3 describes the implementation and Section 3.4 shows the flowchart of the Mod TCLP execution.

3.1 Design of the interface

The generic interface is a set of predicates and its semantics that the constraint solver has to implement. These predicates provide the features that the tabling engine requires in this framework. Fig. 3.1 shows the heads of the predicates required by the Mod TCLP interface and a brief description. The design of this interface is based on three arguments (Dom, ProjStore. Store) and two main operations:

Entailment checks if the constraint store S_a of a new call it is entailed by the projected constraint store S_b of a previous generator. If S_a is entailed by S_b ($\mathbf{S}_a \sqsubseteq \mathbf{S}_b$), any solution in S_a is also a solution of S_b and the call should be suspend. It is performed in a loop calling entail/4 to check entailment with the constraint store of each previous generator.

Projection generates a representation of the projection of the current constraint store S_1 of the call onto a set of variables V. It is performed in two steps: project_domain/2 which returns Dom is called first and then project_gen_store/3, which returns ProjStore, is invoked. The pair (Dom, ProjStore) represents a new constraint store S_2 involving only the variables in V such that: (i) any solution of S_2 is also a solution of S_1 and (ii) a valuation over V which is solution in S_2 is a partial solution of S_1.

project_domain(+Vars, -Dom) Projects the domain of the list of variables Vars in Dom. The term Dom will be used to check entailment. It is a pre-projection of the current constraint store onto Vars.

project_gen_store(+Vars, +Dom, -ProjStore) Returns ProjStore, which completes (with Dom) the representation of the projected constraint store onto the list of variables Vars. Then the current constraint store is updated to be this projection.

project_answer_store(+Vars, +Dom, -ProjStore) As project_gen_store/3 but it does not update the current constraint store.

entail(+Vars$_a$, +Dom$_a$, +Dom$_b$, +ProjStore$_b$) Checks if the call/answer constraint store (Vars$_a$ and Dom$_a$) is entailed by the previous call/answer projected constraint store (Dom$_b$ and ProjStore$_b$).

current_store(-Store) Returns in Store the representation of the current constraint store. It will be used to reinstall Store as the current store later on.

reinstall_store(+Vars, +Dom, +Store) When a generator is complete, reinstall the constraint store represented with Dom and Store onto Vars.

apply_answer(+Vars, +Dom, +ProjStore) Applies the constraints store of the answer represented with Dom and ProjStore to the variables in the list Vars .

Figure 3.1: Generic interface specification.

The predicate project_domain(+Vars, -Dom) is executed before the entailment. It can be used to perform only once actions that entail/4 needs, so they would be repeated in the loop with each previous generator. Dom is used to pass the results of these actions to the entailment phase avoiding their recomputation.

The predicate project_gen_store(+Vars, +Dom, -ProjStore) is executed after the entailment only if the entailment phase fails and the call es marked as generator. With this design, it is possible to postpone the projection operation which could be expensive. This predicate also receives Dom in case the actions executed during the first projection are also needed now.

During the entailment

$$\text{entail}(+\text{Vars}_a,\ +\text{Dom}_a,\ +\text{Dom}_b,\ +\text{ProjStore}_b)$$

the constraint store of the new call is represented by Vars$_a$ and Dom$_a$ and the projected store of the generator is represented by Dom$_b$ and ProjStore$_b$.

3.2 Program Transformation

To execute a program under TCLP in Ciao [14], the programmer has to introduce three directives: (i) to import the tabling package, (ii) to indicate the module that implements the interface, and (iii) to specify which predicates shall be executed with tabling.

```
:- use_package(tabling).
:- const_table_module(...).
:- table dist/3.

edge(a,b,50).
edge(b,a,30).

dist(X,Y,D) :-
        D1 #> 0,
        D2 #> 0,
        D #= D1 + D2,
        dist(X,Z,D1),
        edge(Z,Y,D2).
dist(X,Y,D) :-
        edge(X,Y,D).
```

```
dist(_1,_2,_3) :-
        tabled_call(dist_aux(_1,_2,_3)).

e(a,b,50).
e(b,a,30).

dist_aux(X,Y,D) :-
        D1 #> 0,
        D2 #> 0,
        D #= D1 + D2,
        dist(X,Z,D1),
        edge(Z,Y,D2),
        new_answer.
dist_aux(X,Y,D) :-
        edge(X,Y,D),
        new_answer.
```

Figure 3.2: Transformation of dist/3.

```
tabled_call(Call) :-
 (
  lookup_table(Call, Gen, Vars)
 ;
  save_generator(Call, Gen, Vars)
 ),
 project_domain(Vars, Dom),
 (
  member(Lgen,~l_generators(Gen)),
  entail(Vars, Dom,~dom(Lgen),~prSt(Lgen))->
  suspend_consumer(Call)
 ;
  current_store(Store),
  project_gen_store(Vars, Dom, PrSt),
  save_Lgen(Gen, Lgen, Store, Dom, PrSt),
  push(Lgen),
  execute_generator(Call)
 ),
 consume_answer(Ans,~answers(Lgen)),
 member(Lans,~l_answers(Ans)),
 reinstall_store(~store(Lgen)),
 apply_answer(Vars,~dom(Lans),~prSt(Lans)).
```

```
new_answer :-
 top(Lgen),
 (
  lookup_answer(Lgen, Ans, Vars)
 ;
  save_answer(Lgen, Ans, Vars)
 ),
 project_domain(Vars, Dom),
 (
  member(Lans,~l_answers(Ans)),
  entail(Vars, Dom,~dom(Lans),~prSt(Lans))
 ;
  project_answer_store(Vars, Dom, PrSt),
  save_Lans(Ans, Dom, PrSt),
 ),
 fail.



new_answer :-
 pop(Lgen),
 complete(Lgen).
```

Figure 3.3: Tabled_call/1 and new_answer/0 predicates.
Notation: p(~dom(Lgen)) ≡ dom(Lgen, DomLgen), p(DomLgen).
Instead of ProjStore, PrSt is used to save space.

The transformation applied to the dist/3 program with left-recursion (Fig 2.7) is shown in the Fig 3.2. On the left, the program written in the usual way with directives, and on the right the transformation performed by tabling.

The transformation adds tabled_call/1 to control the execution of the tabled predicates and modifies the original predicate changing the head name and adding the predicate new_answer/0 at the end of the body to collect the answers.

3.3 Implementation

Fig 3.3 shows the implementation of table_call/1 and new_answer/0 in Prolog. In the Ciao system they are actually implemented in C for performance reasons, but here we show a Prolog version to help understanding and for conciseness. This section describes how a program under Mod TCLP

is executed thanks to the program transformation already explained.

The tabling transformation only affects tabled predicates, so we considered that a tabled execution starts with a call C_0 to a tabled predicate. Calling table_call/1 makes the tabling engine take the control of the execution.

The tabling engine uses lookup_table/3 to check if C_0 was already stored in the table or if it has to be saved with save_generator/3 as a new generator. In either case, project_domain/2 projects the domain of Vars (the constrained variables that appear in C_0) in Dom. Then entail/4 is executed against the previous matching calls (Lgen) with its different projected stores. If it is entailed by a previous call C_1 (C_0 is more particular than C_1), then C_0 is suspended with suspend_consumer/1, which protects its memory and stores its trail entries to save them from backtracking. Otherwise, the tabling engine request: the current store Store with current_store/1, and the projection onto Vars with project_gen_store/3.

Then, save_Lgen/5 saves in Lgen the current store (Store) and the projected store (Dom and ProjStore) to protect them from backtracking. The tabling engine stores this information in an implicit stack (push/1) and the generator is executed calling the renamed predicate Call with execute_generator/1. The answers of the generator on the top of the stack will be collected with new_answer/0 and will be used to feed the consumers that entail the projected store of the generator. To guarantee completeness, constraints not belonging to the projected store must not be part of the constraint store during the execution of the generator; if they were present, the set of accepted answers could be reduced and correct answers for a consumer could be discarded.

When the generator produces an answer, lookup_answer/3 checks if it is new and saves it with save_answer/3 otherwise it fails and continues looking for other answers. Then the constraint solver executes project_domain/2 and entail/4 to discard more particular answers and not add them to the answer list. If it is a new answer, project_answer/3 returns in ProjStore a representation of the projected constraint store onto Vars and the answer is saved with save_Lans/3. The predicate new_answer/0 finally fails in order to check all the clauses and to collect all the answers.

A further optimization to reduce the answer list is to check for entailment in the opposite direction and remove previous answers entailed by the new one.

When there are no more clauses to evaluate, the generator executes pop/1 and complete/1. If the generator has suspended consumers, they are restarted. The suspended consumers, are waiting for answers Ans from the generator Lgen in order to restart the execution with consume_answer/2. Then if there is no (more) answers in its generator table, the execution fails and backtracks, otherwise apply_answers/3 applies the answer and checks if the constraint store is still satisfiable and continues the execution, otherwise it retrieves the next answer.

When a generator has no dependence, it is marked complete and calls consume_answer/2. If there are pending answers its original constraint store ProjStore (modified with project_gen_store/3)

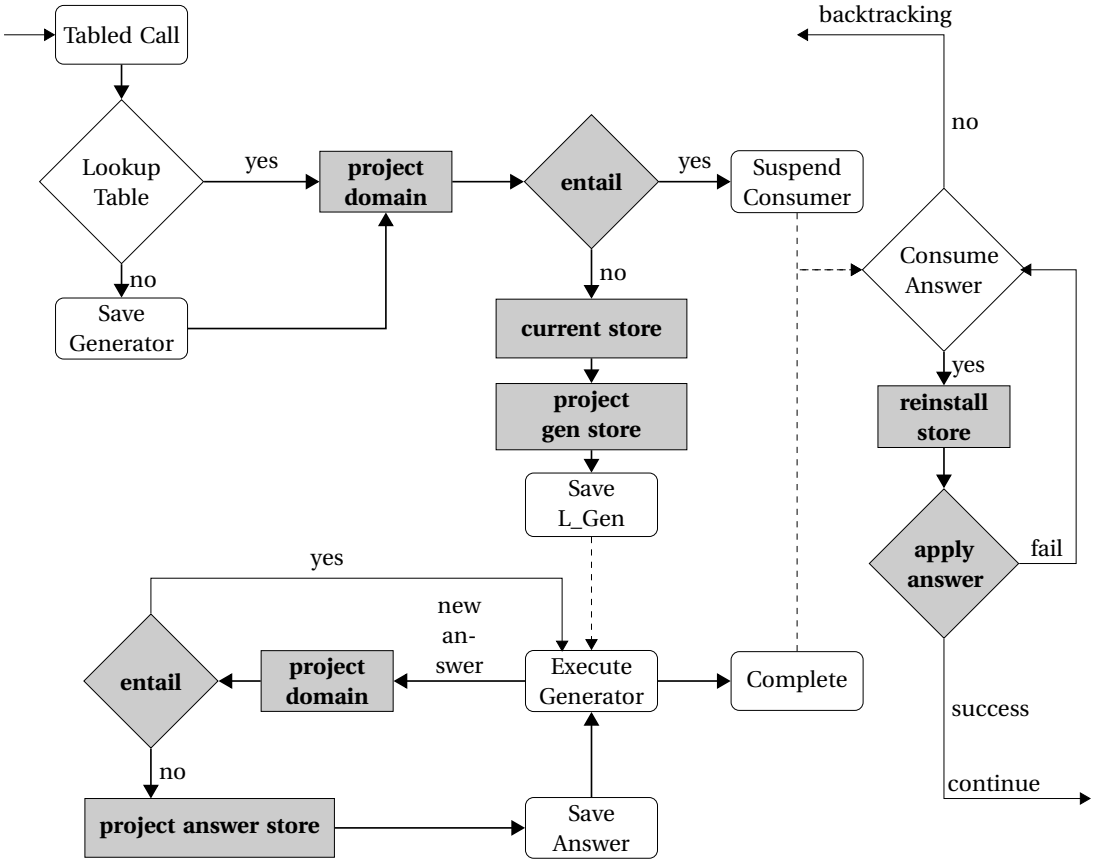

Figure 3.4: Tabled call flowchart with constraint.

has to be restored with reinstall_store/1 and the constraint solver applies its answers as with the consumers.

When the tabled call is a consumer of a generator already complete there is no suspension. The tabling engine returns the answers to the consumer and continues the execution.

3.4 Flowchart (execution of dist/3)

In this section we will use the program in Fig 3.2 and the flowchart in Fig. 3.4 to describe how the query ?- D #< 150, dist(a,Y,D). is executed. This program makes use of CLP(Q); the interface implementation described in Fig. 4.3, and its search tree is shown in Fig. 2.6 (right).

The query starts the tabled execution, since the predicate dist/3 is tabled. The pattern of the query is dist(a,F0,V0) and it is saved as a generator with save_generator/3. Then TCLP(Q) executes project_domain([D], Dom) which succeeds without a value in Dom because entail/4 does not use this argument.

There is no previous generators Lgen with the same call pattern, so the entailment check fails. Then TCLP(Q) executes current_store(Store) which succeeds without returning a value in Store because the constraint store is not modified during the projection. Then project_gen_store([D],

_, _, (F, St) is executed which returns (F, St) = ([V0], [V0 #< 150]), the representation of the projection of the current constraint store {D #< 150} onto D. The tabling engine saves the projected store and starts the execution of the generator. Prolog evaluates the first clause and adds the constraints {D1 #> 0, D2 #> 0, D #= D1 + D2} to the constraint store, Then it calls dist(a,Z,D1) and reenters in the flowchart.

Prolog starts the evaluation of dist(a,Z,D1) calling lookup_table/3 which succeeds because it finds a generator (the previous call) in the table with the same pattern call, and returns in Gen the information of the generator. Then TCLP(Q) checks if this call entails projected constrain store Lgen of this generator calling entail([D1], _, _, ([V0], [V0 #< 150]) which succeeds because the current constraint store {D #< 150, D1 #> 0, D2 #> 0, D #= D1 + D2}[1] is entailed (after the unification [D1] = [V0]) by {V0 #< 150}:

$$\{ D \#< 150, D1 \#> 0, D2 \#> 0, D \#= D1 + D2 \} \sqsubseteq \{ D1 \#< 150 \}$$

This call is marked as a consumer of the generator (its constraint store is more particular than the projected store of the generator) and TCLP(Q) suspends the execution. After backtracking the execution of the generator evaluating the second clause continues. This clause unifies with edge(a,b,50) and the tabling engine collects the first (new) answer Y = b, D = 50 with new_answer/0. Then the tabling engine fails always and by backtracking retrieves the next answer. There are no more clauses to evaluate, so the generator resumes the execution of its consumer. The tabling engine restarts the consumer with the first answer dist(a,b,50) which is applied to the suspended call. This answer satisfies the constraint {D1 #< 149}, so execution continues and Prolog tries to resolve edge(b,Y,D2), which unifies with e(b,a,30) and collects the second answer Y = a, D = 80. Then the consumers of the generator are resumed with this new answer. The answer satisfies the constraints, execution continues and a third answer Y = b, D = 130 is collected. The consumer is resumed again with this new answer but there is no edge clause that satisfy the constraint {D1 #< 20} and it fails. The generator does not have more dependencies and is marked as complete and returns the correct answers of the query.

[1] To clarify, this constraint store es equivalent to {..., D1 #> 0, D1 #< 149, ...}

Chapter 4

Examples of Constraint Solver Integration

With the examples in this chapter we will give an overview of how the interface we propose can be used with different constraint solvers. We have used three solvers with different characteristics in their implementation: a constraint solver for difference constraints (Section 4.1), ported from [4] and completely written in C; the classical implementation of CLP(Q) by Christian Holzbauer, used in many Prolog systems (Section 4.2); and a new solver for a constraint system over finite lattices (Section 4.3), used for an abstract interpreter we use as benchmark (Section 5.3).

4.1 Difference Constraints

The *difference constraint* system $D_{\leq}$ [4] features constraints of the form X - Y $\leq$ d with X,Y,d $\in \mathbb{Z}$, X, Y variables and d a constant. They can be seen as modeling flows in a graph. The constraint store can be represented as an n $\times$ n matrix A of distances, where the distance between v_1 and v_2 is A_{v_1,v_2}. Solvers are based on shortest path algorithms [11], and a constraint store is satisfiable if there are no negative cycles, which can be checked with the Bellman-Ford single-source shortest path algorithm. The projection onto a set of variables V consists of extracting from A a sub-matrix A' for all pairs $\{v_1, v_2\} \subseteq V$. Checking that A entails A' (A' $\sqsubseteq$ A) boils down to checking that $\forall \{v_1, v_2\} \subseteq \mathsf{vars}(A) \cdot A_{v_1,v_2} \leq A'_{v_1,v_2}$.

In our case, the matrix A is implemented in C and the attribute of each constrained variable is its index in the matrix. The Prolog interface of this constraint solver (Fig 4.1) is interesting as it illustrates: (i) how it is possible to plugin external solvers and (ii) how the arguments of the generic interface can be instantiated in a specific way to take into account the characteristics of the constraint solver. Fig 4.2 shows the C function declarations, in the header file (difference_constraint_tab.h, which are directly called from the Mod TCLP interface implemented in Prolog.

Dom The pair (Size,Index) which keeps the size and memory address of the C array with the index of the variables in the matrix.

```
project_domain(Vars, (Size,Index)) :-
        project_domain_c(Vars, Size, Index).
project_gen_store(Vars, (Size, Index), PrSt) :-
        project_gen_store_c(Vars, Size, Index, PrSt).
project_answer_store(Vars, (Size, Index), PrSt) :-
        project_answer_store_c(Vars, Size, Index, PrSt).
entail(_, (SizeA, IndexA), _, PrStB) :-
        entail_c(SizeA, IndexA, PrStB).
current_store(Store) :-
        current_store_c(Store).
reinstall_store(Vars, (Size, Index), Store) :-
        reinstall_store_c(Vars, Size, Index, Store).
apply_answer(Vars, (Size, Index), PrSt) :-
        apply_answer_c(Vars, Size, Index, PrSt).
```

Figure 4.1: Interface for Difference Constraints.

```
void project_domain_c(tagged_t Vars, int Size, int* Index);
void project_gen_store_c(tagged_t Vars, int Size, int* Index, space* PrSt);
void project_answer_store_c(tagged_t Vars, int Size, int* Index, space* PrSt);
void entail_c(int SizeA, int* IndexA, space* PrStB);
void current_store_c(space* Store);
void reinstall_store_c(tagged_t Vars, int Size, int* Index, space* Store);
void apply_answer_c(tagged_t Vars, int Size, int* Index, space* PrSt);
```

Figure 4.2: C function declaration in difference_constraint_tab.h.

ProjStore The memory address of a new C sub-matrix from the current store containing only the entries whose indexes are in Dom.

Store The memory address of the copy of the C matrix that represent the current constraint store.

The definition of these arguments comes from the entailment definition:

entail(_, (Size_a, Index_a), _, ProjStore_b) :- ...

The predicate project_domain(Vars, (Size, Index)) accesses the attributes of the variables in the list Vars, saves them in a C array and returns its size in Size and the memory address of the array in Index. This predicate is invoked before the entailment check only once, but without Dom during the entailment, the solver should have to access the attributes of the variables, for each Lgen and create the array with the indexes which would lean extra overhead. The indexes which represent the variables of the call in the projected matrix from the generator, are used to know the columns and rows in the current matrix

4.2 CLP(Q)

Holzbauer's CLP(Q) [15, 16] is a Prolog extension for Constraint Logic Programming over the rationals. The implementation in Ciao solves linear equations over rationals providing valued variables, covers the lazy treatment of non-linear equations, features a decision algorithm for lineal inequalities that detects implied equations, removes redundancies, performs projections (quantifier elimination) and allows for linear dis-equations.

The interface for this constraint solver (Fig. 4.3) highlights that Mod TCLP can be implemented

```
project_domain(_, _).
project_gen_store(V, _, (F, St)) :-
      clpqr_dump_constraints(V, F, St).
project_answer_store(V, _, (F, St)) :-
      clpqr_dump_constraints(V, F, St).
entail(V, _, _, (V, St)) :-
      clpq_entailed(St).
current_store(_).
reinstall_store(_, _, _).
apply_answer(V, _, (V, St)) :-
      clp_meta(St).
```

Figure 4.3: Interface for CLP(Q).

```
project_domain(Vars, [Domain, (Vars, Space)]) :-
   get_domain(Vars, Domain),
   dump_constraint(Vars, (Vars, Space)).
project_gen_store(Vars, [_, (Fresh, Space)], _) :-
   clean_constraints(Vars),
   Vars = Fresh,
   call_constraints(Space)
project_answer_store(_, _, _).
entail(_, [DomainA, (FreshA, SpaceA)], [DomainB, (FreshB, SpaceB)], _) :-
   entail_domain(DomainA, DomainB),
   FreshA = FreshB,
   subtract(SpaceA, SpaceB, []).
current_store(_).
reinstall_store(_, _, _).
apply_answer(Vars, [Domain, (Fresh, Space)], _) :-
   apply_domain(Vars, Domain),
   Vars = Fresh,
   call_constraints(Space).

entail_domain([], []).
entail_domain([X|Xs], [Y|Ys]) :-
   abs_meet(X, Y, X),
   entail_domain(Xs, Ys).
```

Figure 4.4: Interface for constraint over (finite) lattices.

with only three operations: entailment, projection and application of answers.

The existing CLP(Q) predicates already provide the functionality necessary by our tabling engine; therefore we only have to write bridge predicates. In this constraint solver only the argument ProjStore = (F, St) is needed, where: (i) F is a copy of the list of constrainted variables (V) with fresh variables and (ii) St is a list with the projection of the constraint store onto V referred to the variables in F.When an argument is not needed an anonymous variable (_) is used in the interface.

The tabling engine stores the constraint arguments (F, St) to protect them from backtracking, so when the tabling engines gives back this arguments to the solver they will be fresh in any case. So it is possible to optimize the projection removing the generation of fresh variables.

The argument Store is not used in this interface because CLP(Q) is implemented in Prolog and during tabled execution, reinstalling the constraint store is performed transparently by the tabling engine.

4.3 Constraints over (Finite) Lattices

A lattice is a non-empty ordered set S with two operations, *join* ($x \wedge y = \sup\{x,y\}$) and *meet* ($x \vee y = \inf\{x,y\}$) such that $\forall\ x,y \in S$ there exists $x \wedge y$ and $x \vee y$. The only basic constraint we require from a finite lattice is $x \sqsubseteq y$, meaning that $x \wedge y = y$. This is defined solely based on the topology of the lattice, and does not require knowledge about the semantics of the points in the lattice or the operations defined among them. This leads to a two-layer implementation of the lattice: one which deals with the topological relation between its points and another one which deals with the operations among these elements.

The constraint over lattices solver implements: (i) a general constraint predicate $\sqsubseteq$ common to all the lattice (the solver knows its properties and how to make the projection) and (ii) an algorithm to propagate the constraint between variables without knowing of the semantic of the operations defined in the lattice domain.

Fig 4.4 shows an expanded version of the interface implemented for the constraint over (finite) lattice solver in order to illustrate the main operations performed to provide the features required by the tabling engine. The implementation of the interface is in Annex A.5.5 and the implementation of the constraint solver is in Annex A.5.3.

To evaluate the entailment, the lattice solver requires the projection of the domain and the constraint store from the new call and from the generator. Prolog takes care of the constraint store of the generators thanks to backtracking, so the argument Store is not used. The solver interface only makes use of the argument Dom = [Domain, (Vars, Space)], where: (i) Domain isa list with the abstract domain of the variables (in the same order) and (ii) Vars is the list with the constrained variables and Space is a list with the constraint store projected onto Vars. The entailment operation

```
entail(_, [DomainA, (FreshA, SpaceA)], [DomainB, (FreshB, SpaceB)], _) :- ...
```

receives the arguments with fresh variables and performs two steps: first, it checks that each abstract domain in DomainA is included (in the sense of $\sqsubseteq$) in its respective abstract domain in DomainB, and failing, the entailment fails; and second, it checks that every constraint in SpaceA is also in SpaceB, if so, the set of possible valuations of the variables in [DomainA, [FreshA, SpaceA)] is a subset of the corresponding valuations in [DomainB, (FreshB, SpaceB)]. Otherwise the entailment fails.

Although the argument ProjStore is not used, the predicate

```
project_gen_store(Vars, [_, (Fresh,Space)], _) :- ...
```

has to be executed to remove the constraints not projected in Space from the current store. This operation removes the un-projected lattice related constraints.

As mentioned before, the constraint solver is implemented in two layers: (i) the constraint lattice solver and (ii) the lattice-domain which defines the elements of the partial order set and its

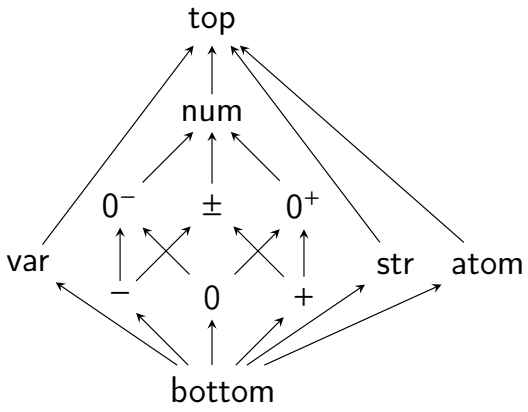

Figure 4.5: Abstract domain.

operations.

The modularity of the proposal architecture makes it possible to seamlessly use different abstract domains. Fig 4.5, shows the sign abstract domain that the constraint solver imports to execute the operations in it. Module abstraction.pl implements the sign abstract domain and its code is in Annex A.5.4. This domain will be used in the implementation of the abstract interpreter (section 5.3).

The abstract values are the elements of the lattice, and the edges in the figure represent the $\sqsubseteq$ constraint (e.g. pos $\sqsubseteq$ pos+z). The operations (e.g. X is Y + Z) and relations (e.g. X < Y) between the elements (abstract values) are defined in abstraction.pl using truth tables (see Annex A.5.4). The abstract domain is an abstraction of real values in a finite set of elements and the operations reproduce the behavior of these operations in Prolog. The goal X is 3 * (-1) in the abstract domain is evaluated as X~var is Y~pos * Z~neg where the notation V~v means that the concrete variable V is assigned the abstract value v in the interpreter. After abstract execution of this goal the abstraction of X is neg, because it is the success abstract value of X for this goal.

To illustrate how the interface works, we use as an example three variables [X,Y,Z] which at some point of the execution could have these constraints:

X~neg, Y~neg+z, Z~num, X $\sqsubseteq$ Y, X < Z, Y $\sqsubseteq$ Z

When the tabling engine with this constraints calls

project_domain([X, Z], [Domain, (Fresh, Space)]).

the solver calls: (i) get_domain(Vars, Domain) and returns Domain = [neg, num]. and (ii) dump_constraintVars, (Vars, Space) that uses the Fourier's algorithm (see [21] p.55-57) for variable elimination to project the conjunction of the $\sqsubseteq$ constraints (in) onto X and Z, and returns (Vars, Space) = ([X,Z], [X in Z]). Finally the solver returns to the tabling engine:

Dom = [[neg, num], ([X, Z], [X in Z])]

As we said, project_gen_store(Vars, [_, (Fresh, Space)], _) is used to remove the constraints regarding the lattice domain, since the solver does not project them. The arguments received from the tabling engine (Fresh, Space) are fresh variables so the unification Vars = Fresh restores the

correct renaming of the constraint. Since the constraint solver is implemented in Prolog, it does not need to stash away the current constraint store because the system, after backtracking and resuming the generator, will undo the clean_constraints(Vars) operation and the current constraint store will be "re-installed" for free.

After the domain projection, the exectuion continues with the entailment phase which is performed in two steps: (i) entail_domain(DomainA, DomainB) checks that for each $x_i \in$ DomainA from the new call and each $y_i \in$ DomainB from the generator $x_i \sqsubseteq y_i$, (otherwise the entailment fails), and (ii) the solver with subtract(SpaceA, SpaceB, []), unifies the "fresh" lists, and checks that all the constraints in the constraint store of the generator (Space_b) are also in the constraint store of the call (Space_a). If so that means that also Space_a $\sqsubseteq$ Space_b and the entailment succeeded.

The last operation is the application of the answers with

apply_answer(Vars, [Domain, (Fresh, Space)], _)

which is executed in two steps as well: (i) the solver first checks with apply_domain(Vars, Domain) that for each variable in Vars the corresponding abstract value in Domain is below or equals ($\sqsubseteq$) to the current abstract value (it fails otherwise), and (ii) then, after the unification of the lists Fresh and Vars, the solver uses call_constraint(Space) to execute the constraints of the answer's projected constraint store Space. If the current constraint store is still consistent, the execution continues, otherwise it fails and backtracks.

Chapter 5

Experimental Evaluation

This chapter shows the evaluation of the performance of the Mod TCLP framework using the constraint solvers presented in Section 4. The use cases, together with a full Ciao Prolog distribution including the libraries and interface presented in this thesis, are available at `http://goo.gl/vWRV15`. All benchmarks in the present thesis were executed using that distribution on a Mac OS-X 10.9.5 machine with a 2,66 GHz Intel Core 2 Duo processor. All times are given in milliseconds.

In general, the adoption of a more modular framework for interfacing constraint systems and a tabling engine incurs an additional overhead. This is in a great part due to the need to pass information from the tabling engine (e.g., C) level to the interface (Prolog) level. There is quite some room for optimization at the architectural and coding level, and we are confident that we will be able to improve the presented results. On the other hand, and in exchange for this performance penalty, we gain a great deal of flexibility when adding more constraint systems which, after the initial steep curve when designing and implementing the first working version of the interface, proved to be quite easy (see the interface for CLP(Q) in Section 4.2, for example).

5.1 TCLP vs Mod TCLP

In order to evaluate the costs of adopting a more modular framework we compare Mod TCLP(diff) against the implementation of TCLP(diff) [4] using: **truckload(P, Load, Dest, Time)** (which solves a shipment problem; in the examples, $\mathsf{P} = 30$, $\mathsf{Dest} = \mathsf{Chicago}$ and the Load varies for every invocation, its code is in Annex A.1), and **path(L)** (a left-recursive graph reachability program where L is the total number edge traversals, which is constrained to ensure execution finishes, its code is in Annex A.2), taken from [26] and also used in [4] to evaluate TCLP(diff).

Table 5.1 shows that the use of the new modular framework (Mod TCLP) increases nearly five-fold the execution time with respect to the previous TCLP, but it is still competitive if we compare it with

	CLP	TCLP	Mod TCLP
truckload(300)	41950	2791	11150
truckload(200)	4276	952	3185
truckload(100)	140	129	221

	TCLP	Mod TCLP
path(30)	3206	15159
path(20)	2700	12568
path(10)	1053	4436

Table 5.1: Run time results for the truckload/4 and path/1.

a CLP execution. This is mainly due to the fact that the execution control passes from the tabling engine (in C) level to the interface level (in Prolog) which calls back the constraint solver (in C). In the implementation of TCLP, the communication between the tabling engine and difference constraints was however made at the C level. As mentioned before, there is quite some room for optimization at the architectural and coding level, and we are confident that we will be able to improve the presented results.

5.2 Difference Constraints vs CLP(Q)

We used the previous benchmark to compare CLP(Diff) with CLP(Q) and we observed that CLP(Q) was 5 times slower. When we compared TCLP(Diff) with Mod TCLP(Q) we observed that Mod TCLP(Q) is 30 times slower. This overhead is due to: (i) CLP(Q) being complex than Difference constraints and implemented in Prolog, (ii) the projection and entailment in TCLP(Q) being more expensive than in TCLP(Diff), and (iii) the overhead imposed by the modular framework that we evaluated in the previous section.

In this section we will show the benefits of modularity in terms of the expressiveness that CLP(Q) provides, which can be used not only to write more readable programs, but also to increase the performance.

To evaluate this, we use the fibo/2 program (Code in Annex A.3), the well-known doubly recursive Fibonacci function, using CLP(Diff) and CLP(Q). The code for these two constraint domains is sloghtly different because difference constraints can only have two variables per constraint; therefore the constraint F #= F1 + F2 before the recursive calls is not valid, and has to be moved after the calls. Fig. 5.1 shows the fragments which differ in the two versions.

The results presented in Table 5.2, which we call reverse Fibonacci (to find N given the N^{th} Fibonacci number), illustrate that in this case the added power of CLP(Q) pays off, as the constraints can be added early in the execution.

```
...                              ...
fib(N1, F1),                     F #= F1 + F2,
fib(N2, F2),                     fib(N1, F1),
F #= F1 + F2.                    fib(N2, F2).
```

Figure 5.1: Fragment of two versions of fibonacci/2: diff. constraints (left) vs. CLP(Q) (right).

	Diff Constraints			CLP(Q)	
	CLP	TCLP	Mod TCLP	CLP	Mod TCLP
fibonacci(P, 89)	494	11	13	67	12
fibonacci(P, 610)	–	21	25	628	20
fibonacci(P, 4181)	–	36	42	6001	30
fibonacci(P, 28657)	–	56	69	153813	40
fibonacci(P, 196418)	–	85	111	> 5 min.	53
fibonacci(P, 832040)	–	113	158	> 5 min.	64

Table 5.2: Run time results for the fibonacci/2 program in two versions.

The next benchmark evaluates the performance of the distance-bounded reachable nodes problem (Code in Annex A.4), which we already described in chapter 2, we uses to sources of variation to explore different scenarios: on one hand, using a graph with or without cycles, and on the other hand writing the dist/3 predicate in four semantically equivalent ways, namely:

With the recursive clause before the base clause:

ep_e/3 using right recursion (Fig. 2.5).
pe_p/3 using left recursion (Fig. 2.7).

With the recursive clause after the base clause:

e_ep/3 using right recursion.
e_pe/3 using left recursion.

The left-recursive versions do not terminate in CLP(Q) due to the lack of entailment and suspension, but as we said in chapter 2.3. The execution of the right-recursive versions finish in CLP(Q) with the cycled graph since the calls have a bound on the length of the path. As table 5.3 shows, the left recursive version is always faster with Mod TCLP(Q), and the right-recursive one is only slightly slower in Mod TCLP(Q) the case of not having cycles.

	Without cycles		With cycles	
	CLP(Q)	Mod TCLP(Q)	CLP(Q	Mod TCLP(Q)
ep_e	210	235	495	178
pe_e	–	42	–	85
e_ep	213	237	497	192
e_pe	–	42	–	85

Table 5.3: Run time results for the path_distances program (in the four versions).
A '–' means that the query does not terminate.

5.3 An Abstract Interpreter with Tabling and Constraints

We evaluates now the applicability of our framework in an abstract interpreter [7,31] (Code in Annex A.5), implemented in two layers: one layer takes care of the fix-point and the other uses the

absint.pl This module implements the fix-point algorithm using tabling. The predicate analyze/2 is the entry point to analyze the programs. The tabled predicate step_goal_att/1 guarantees that the interpreter reaches a fix-point.

stored_program.pl This file stores as a fact the programs to be analyzed using the syntax stored_program(head(A1, ... , An), [goal_1, ... , goal_n]).

lattice_solver.pl The constraint solver can be used with two variants: with the lattice and domain operations written either in a *functional* style (e.g., inputs to abstract domain operations produce outputs) or as a constraint system, which makes it possible to issue richer queries. This file implements the lattice operations, the bridge with the abstract domain operations, and the predicates required by the interface Mod TCLP.

abstraction.pl This module implements the sign abstract domain operations. The use of truth_table/1 for declarativeness reduces the performance, but the use of tabling to memorize the calls to calculate/4 alleviates this overhead.

lattice_solver_tab Since the predicates required by the projection, entailment, and application operations are defined in the constraint solver, the interface is a bridge which execute, the required predicates from the constraint solver.

Figure 5.2: Description of the abstract interpreter implementation.

constraint operations to perform the abstract evaluation. As we described in chapter 4.3 the constraint solver it is also implemented in two layers: the lattice constraint solver, which performs the constraint operations, and the abstract domain which defines the abstraction and the operations. Fig 5.2 describes the files which implemented the abstract interpreter.

To validate the benefits of the use of constraint solving in the abstract interpreter, we analyze two variants (permute/n and permute_p/4) of a synthetic program presented in [12] and compare its performance against an implementation which only use tabling.

5.3.1 Analysis of permute/n.

In this example (see Fig. 5.3), the permutation in the order of the arguments in the recursive clauses generates many different abstract substitutions which make it possible to exercise the different mechanisms in tabling. The program permute/n, is parametrized by n the number of arguments.

```
permute(A1, A2, ... , An) :- A1 < 3, permute(A2, ... , An, A1).
permute(A1, A2, ... , An) :- A1 < 0, permute(A2, ... , An, A1).
permute(A1, A2, ... , An) :- A1 = -3, A2 = -3, ... , An = -3.
```

Figure 5.3: Permute/n program to be analyzed.

The query ?- analyze(permute(A1, ... , An), P). is used to analyze the program permute/n and returns in P a list with the abstract domain of the variables when the program succeeds. As we said, we analyze the program with to version of the abstract interpreter:

- **The tabled version:** This version does not use the constraint solver. During the analysis of the program, the analyzer checks the goals of each clause with the call step_goal_table(Goal, InfoIn, InfoOut. This predicate is tabled to avoid entering a loop and guarantees that a fix-point is reached (Fig. 5.4 on the left shows its code). The arguments of this predicate are: Goal, which identifies the goal to be analyzed; InfoIn, which is a list with the abstract value of the arguments of Goal when it is called, and InfoOut which is another list with the abstract value of the variables when Goal succeeds.

  The execution of the query to analyze the program initializes the variables to the abstract domain top and then executes the call step_goal_table(permute(A1, A2, A3, A4), [top,top,top,top], InfoOut). Then the goal $A1 < 3$ is interpreted and the abstract values of the variables in the call become [num,top,top,top]. Then the interpreter has to analyze the goal permute(A2,A3,A4,A1). The analyzer repeat the call step_goal_table(permute(A2, A3, A4, A1), [top,top,top,num], InfoOut). This executions under SLD means that the analyzer enter in loop, but the use of tabling guarantees that the analysis finishes because the answer space is finite.

  The new call is not a variant of the previous one (num is not equal to top) and this call is marked as generator and execution continues. With tabling the analyzer repeats the call to analyze this predicate four times with different patterns and finally suspends with step_goal_table(permute(A1, A2, A3, A4), [num,num,num,num], InfoOut). The evaluation of the second clause produces also a permutation so it is executed another four times and suspend with step_goal_table(permute(A1, A2, A3, A4), [neg,neg,neg,neg], InfoOut) and then succeeds with the last clause. With this answer the tabling engine resumes the exectution of the first consumer and so on, until all the combinations are evaluated and the tabling engine reach the fix point.

- **TCLP version:** This version uses the constraint over (finite) lattices to handle the domains and solve the constraints over the variables. In this versions, the tabled call to analyze the first goal is step_goal_att(permute(A1, A2, A3, A4). The information about the domain of the variables is part of the constraint store.

  This first call is stored as a generator with the pattern permute(V_0,V_1,V_2,V_3) and its projected domain is ([top,top,top,top], ([A1, A2, A3, A4],[])). Then it is resolved the first goal of the first clause of the program, $A1 < 3$. This goal modifies the domain of A1 and the solver updates the constraint store. Then the analyzer calls step_goal_att(permute(A2, A3, A4, A1) whose pattern matches with the initial call. Then the solver projects the domain ([top,top,top,num], ([A2, A3, A4, A1],[])) and starts the entailment phase calling

```
:- table step_goal_table/3.

step_goal_table(Goal, InfoIn, InfoOut) :-
    stored_program(Goal, Body),
    interp_body(Body, InfoIn, InfoOut).
```

```
:- table step_goal_att/1.

step_goal_att(Goal) :-
      stored_program(Goal, Body),
      interp_body_att(Body).
```

Figure 5.4: Step_goal_table on the left and step_goal_att on the right.

	Tabling	Mod TCLP
permute/10	2788	3
permute/8	563	2
permute/6	112	2
permute/4	21	1

Table 5.4: Run time results for ?- analyze(permute(A1, ... , An), P).

entail(_, ([top,top,top,num], (Vars,[])), ([top,top,top,top], (Vars,[])), _).

The entailment succeeds because top $\sqsubseteq$ top and num $\sqsubseteq$ top and there are no constraints to be checked. The tabling engine then marks the call as consumer and suspends it execution. The generator continues the evaluation with the second clause and resolves the first goal of the second clause of the program, A1 $<$ 0. This goal modifies the domain of A1 and the solver updates the constraint store. Then the analyzer calls step_goal_att(permute(A2, A3, A4, A1) which also matches with the initial generator. Then the solver projects the domain ([top,top,top,neg], ([A2, A3, A4, A1],[]) and checks entailment. The constraint store of this new call is also entailed by the constraint store of the generator (neg $\sqsubseteq$ top), so the call is termed as consumer and its execution is suspended.

The generator continues the execution with the last clause and finds an answer. Then the solver projects the answer's domain Dom = ([neg,neg,neg,neg], (Vars, [])) and the tabling engines saves it in the table. Then the consumers are restarted with this answer. In both cases the answer satisfies the constraint solver but no new answers are generated, so the execution fails. The generator does not have more dependencies and is marked as complete and returns the answer of the query:

P = [A1 in neg, A2 in neg, A3 in neg, A4 in neg]

Table 5.4 shows that Mod TCLP is considerably faster because it reduces the search tree using the lattice topology of the abstract domain. On the other hand the tabled version has to analyze all the permutations, increasing the execution time needed to find the correct answer.

5.3.2 Analysis of permute_p/4.

With this example (see Fig. 5.5), we want to evaluate the effect of the constraint propagation in the comparative between the two implementation of the abstract interpreter. The implementation without the constraint solver should be in principle faster but less precise because it does not perform the propagation which is expensive. And the implementation with the constraint solver that propagate the relations among variables, resuming the stored constraints and immediately updating the domains when a domain changes, should be more precise.

We are going to use three different analysis scenarios to evaluate the impact of propagation in accuracy and performance.

```
permute_p(X1,X2,X3,X4) :- X1 > 0, permute_q(X1,X2,X3,X4).
permute_p(X1,X2,X3,X4) :- X1 = 0, permute_q(X1,X2,X3,X4).
permute_p(X1,X2,X3,X4) :- X1 < 0, permute_q(X1,X2,X3,X4).
permute_q(X1,X2,X3,X4) :- permute_p(X2,X3,X4,X1).
permute_q(X1,X2,X3,X4) :- X1 < 0, permute_q(X2,X2,X3,X4).
permute_q(X1,X2,X3,X4) :- X1 < X2, X2 < X3, X2 < X4.
```

Figure 5.5: Permute_p program to be analyzer.

	Tabling w.o. constraints	Mod TCLP	More Precise
Without restrictions	154	88	Mod TCLP
Restrictions after	154	89	Mod TCLP
Restrictions before	121	19	Mod TCLP

Table 5.5: Run time results for the analyze/2 program.

First, we perform a query which analyzes permute_q/4 without imposing constraints on the program variables. In this case the Mod TCLP version returns a more precise answer, in the analysis results corresponding to Mod TCLP all the variables have domains which are more concrete (in the lattice) than those for the tabling without constraints.

```
?- analyze(permute_p(A1,A2,A3,A4),P).

     P = [A1 in num,A2 in top,A3 in top,A4 in top] tabling w.o. constraints
     P = [A1 in num,A2 in num,A3 in num,A4 in num] Mod TCLP
```

In the second example the query filters the answers adding some constraints onto the variables using the functional properties of the constraint solver. The execution with the tabled version does not benefit from the propagation of the constraints, so A1 and A2 are not updated due to the constraint over A3. In this example the answer from Mod TCLP is more precise.

```
?- analyze(permute_p(A1,A2,A3,A4),P), A1 in A2, A2 in A3, A4 in pos+z, A3 in A4.

     P = [A1 in num  ,A2 in top  ,A3 in pos+z,A4 in pos+z] tabling w.o. constraints
     P = [A1 in pos+z,A2 in pos+z,A3 in pos+z,A4 in pos+z] Mod TCLP
```

In this last example, the constraint-based implementation makes it possible to check properties by *setting* them beforehand and then calling the analyzer:

```
?- A1 in A2, A2 in A3, A4, pos+z, A3 in A4, analyze(permute_(A1,A2,A3,A4),P).

     P = [A1 in num,A2 in top,A3 in pos+z,A4 in pos+z] tabling w.o. constraints
     P = [A1 in pos,A2 in pos,A3 in pos  ,A4 in pos  ] Mod TCLP
```

As we commented there are no benefits from propagation in the tabled version (the solution does not change), but with Mod TCLP the final solution is more precise due to the propagation of the constraints during the analysis of the program: when the variable domains are not consistent with the constraints, the analysis fails and a bottom abstract value is generated.

Table 5.5 shows that the constraint-based implementation is somewhat faster when analyzing programs which do not have constraints previously set, or when properties are checked after the analysis, and much faster when the properties to check are set before the analysis. In this case, only

the constraint-based implementation takes full advantage of the entailment to reduce the search tree. In all cases, the constraint-enhanced domain is more precise, as we explained, thanks to the propagation. To summarize, the abstract interpreter implemented using Mod TCLP:

- Is more precise because the solver keeps the relations between variables and when their domains change, propagation takes place and the domain of the variables is updated to satisfy the constraints.
- Increases the performance because it uses the projection of the constraint in (using the Fourier algorithm) and performs entailment when for all the variables of a call, their abstract values are part of ($\sqsubseteq$ the abstract values of the variables of the generator and when the constraint store of the call is more particular than the projected store of the generator.

Chapter 6

Conclusions and Future Work

We have presented an approach to include constraint solvers in logic programming systems with tabling. Our main goal is to make adding additional constraint solvers easier; this was a non-trivial task in previously existing systems. In order to achieve this, we determine the *services* that a constraint solver has to offer to a tabling engine. These services are abstract in the sense that the constraint solver is free to decide how to implement them, and have been designed to cover many possible implementations.

To validate our design we have interfaced one solver previously written in C (difference constraints), an existing, classical solver (CLP(Q)) and a new solver (constraints over finite lattices), and we have found the integration to be easy — certainly easier than with other designs, given the capabilities that our system provides. In some cases, some of these services can be void, since the implementation of the solver does not need to execute the actions associated with them by default.

We also evaluated the performance for a series of benchmarks. In some of them large savings are attained w.r.t. non-tabled execution — even taking into account the penalty to pay for the additional flexibility and modularity. We are in any case confident that there is still ample space to improve the efficiency of the implementation, since we are presenting an initial prototype in which we internally gave more importance to the cleanliness of the code and the design.

Future work could focus on the improve of the Mod TCLP interface and it application to intelligent management of big data. Besides the work in traditional SQL (and also non-SQL) databases, there are approaches in the literature related to our proposal, namely those coming from the Datalog and deductive databases community. These are now being applied to big data management (e.g., the work by Carlo Zaniolo from UCLA [34], the products by the U.S.A. company LogicBlox [20], and also the works of XSB Inc [25]).

However, existing approaches can fall short in several aspects in the realm of big data management. On one hand, making complex reasoning over (complex) data needs complex queries,

which are hard to write in SQL; the same applies to the map-reduce approaches used in cloud environments, up to the point that it may be difficult to really understand or prove that the query is actually representing the kind of reasoning that one wants to make. A higher-level language (e.g., featuring algebraic data structures and constraints), closer to knowledge representation, would be of great help here. Logic programming + constraints offer just these facilities, much beyond what SQL and Datalog can offer, and much more concisely than what map/reduce offers.

In addition, current approaches have to be explicitly engineered to take into account incremental data, such as incoming data streams from, for example, time-stamped weather data, traffic monitoring, or even logs of cloud-related activities if threats have to be actively monitored. We intend to make treating this case a native capability of our system (therefore enabling non-monotonic reasoning) by using, on one hand, the monotonicity of logic programming w.r.t. additions to the fact database (i.e., registries) and introducing the revision of previous inferences when facts are removed, which is a form of non-monotonicity.

We envision advantages in several fronts: complex queries and non-trivial reasoning will be easier to express thanks to the higher-level of logic programming and constraints; less computations will be necessary thanks to the automatic reuse of previous inferences brought by tabling (which in some sense performs dynamic programming in an automatic way); queries and associated actions (if any) can be programmed using the same formalism; and large-scale parallelism can be exploited in cloud environments thanks to the referential transparency and declarative nature of the language.

Bibliography

[1] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] Johan Bengtsson and Wang Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124, 2003.

[3] Christian Bessiere. Constraint propagation. *Handbook of constraint programming*, pages 29–83, 2006.

[4] P. Chico de Guzmán, M. Carro, M. Hermenegildo, and P. Stuckey. A General Implementation Framework for Tabled CLP. In Tom Schrijvers and Peter Thiemann, editors, *FLOPS'12*, number 7294 in LNCS, pages 104–119. Springer Verlag, May 2012.

[5] P. Codognet. A Tabulation Method for Constraint Logic Programming. In *INAP'95*, Tokyo, Japan, Oct. 1995.

[6] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *J. Log. Program.*, 27(3):185–226, 1996.

[7] P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2-3):103–179, 1992.

[8] Baoqiu Cui and David Scott Warren. A System for Tabled Constraint Logic Programming. In *Computational Logic*, pages 478–492, 2000.

[9] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems – A Case Study. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 117–126, New York, USA, 1996. ACM Press.

[10] Rina Dechter. *Constraint Processing*. Morgan Kauffman Publishers, 2003.

[11] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully Dynamic Shortest Paths and Negative Cycles Detection on Digraphs with Arbitrary Arc Weights. In *ESA*, pages 320–331, 1998.

[12] Samir Genaim, Michael Codish, and Jacob Howe. Worst-Case Groundness Analysis Using Definite Boolean Functions. *Theory and Practice of Logic Programming*, 1(05):611–615, 2001.

[13] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.

[14] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *TPLP*, 12(1–2):219–252, 2012. http://arxiv.org/abs/1102.5497.

[15] C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *1992 International Symposium on Programming Language Implementation and Logic Programming*, pages 260–268. LNCS 631, Springer Verlag, August 1992.

[16] C. Holzbaur. OFAI clp(q,r) manual, edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.

[17] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[18] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint Query Languages. *J. Comput. Syst. Sci.*, 51(1):26–52, 1995.

[19] J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[20] LogicBlox. `http://www.logicblox.com/`.

[21] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.

[22] A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[23] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, and D.S. Warren. Efficient Model Checking Using Tabled Resolution. In *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 143–154. Springer Verlag, 1997.

[24] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Access Mechanisms for Tabled Logic Programs. *Logic Programing*, 1(38):31–54, 1999.

[25] K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *ACM SIGMOD International Conference on the Management of Data*, pages 442–453. ACM Press, 1994.

[26] Tom Schrijvers, Bart Demoen, and David Scott Warren. TCHR: a Framework for Tabled CLP. *TPLP*, 8(4):491–526, 2008.

[27] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[28] Terrance Swift and David Scott Warren. XSB: Extending Prolog with Tabled Logic Programming. *TPLP*, 12(1-2):157–187, 2012.

[29] H. Tamaki and M. Sato. OLD Resolution with Tabulation. In *Third International Conference on Logic Programming*, pages 84–98, London, 1986. Lecture Notes in Computer Science, Springer-Verlag.

[30] David Toman. Constraint Databases and Program Analysis Using Abstract Interpretation. In *CDTA*, volume 1191 of *LNCS*, pages 246–262, 1997.

[31] David Toman. Memoing Evaluation for Constraint Extensions of Datalog. *Constraints*, 2(3/4):337–359, 1997.

[32] D. S. Warren. Memoing for Logic Programs. *Communications of the ACM*, 35(3):93–111, 1992.

[33] R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.

[34] Carlo Zaniolo. A logic-based language for data streams. In *SEBD*, pages 59–66, 2012.

[35] Youyong Zou, Tim Finin, and Harry Chen. F-OWL: An Inference Engine for Semantic Web. In *Formal Approaches to Agent-Based Systems*, volume 3228 of *Lecture Notes in Computer Science*, pages 238–248. Springer Verlag, January 2005.

Appendix A

Additional Information

A.1 Shipment Problem

truckload(P, Load, Dest, Time): Solve a shipment problem. Determine whether a subset of the packages numbered 1 to P can fill up a truck with a maximum Load to be shipped to Dest. Time is captured as the bounds of the constrained variable Time. In the examples, P = 30, Dest = Chicago and the Load varies for every invocation.

```
%% USER MANUAL
%% - truckload(I,W,D,T): truckload problem where I is the number of packets,
%%   W is the load, D is the final city and T is the time restrictions.
%% - call examples:
%%    truckload(60,100,chicago,T)
%%    truckload(60,200,chicago,T)
%%    truckload(60,300,chicago,T)

:- module(truckload,
        [
            truckload/4
        ],[]).

:- use_package(library(tabling)).
:- const_table_module(library(difference_constraints/difference_constraints_tab)).
:- table truckload/4.

:- use_package(library(difference_constraints)).

truckload(0,0,_,_).
truckload(I,W,D,T) :-
        I > 0,
        I1 is I - 1,
        truckload(I1,W,D,T).
truckload(I,W,D,T) :-
        I > 0,
        pack(I,Wi,D,T),
        W1 is W - Wi,
        W1 >= 0,
        I1 is I - 1,
        truckload(I1,W1,D,T).
```

```
pack(60,29,chicago,T) :- T #>= 19, T #=< 30.
pack(59,82,chicago,T) :- T #>= 20, T #=< 30.
pack(58,24,chicago,T) :- T #>= 8,  T #=< 13.
pack(57,11,chicago,T) :- T #>= 21, T #=< 23.
pack(56,57,chicago,T) :- T #>= 8,  T #=< 29.
pack(55,30,chicago,T) :- T #>= 14, T #=< 19.
...
% there are 60 diferent pack numbered from 1 to 60
...
pack(3, 60,chicago,T) :- T #>= 4,  T #=< 29.
pack(2, 82,chicago,T) :- T #>= 28, T #=< 29.
pack(1, 41,chicago,T) :- T #>= 27, T #=< 28.
```

Figure A.1: Code of the truckload/4 program.

A.2 Node Reachability

path(L) is a left-recursive graph reachability program where L is the total number edge traversals, which is constrained to ensure execution finishes. It is run on a graph with 30 nodes and 742 edges.

```
%% USER MANUAL
%% - path(N) executes an standard query (N is the maximun path length)

:- module(path_dc,
        [
            path/1,
        ],[]).

:- use_package(library(tabling)).
:- const_table_module(library(difference_constraints/difference_constraints_tab)).
:- table reach/4.

:- use_package(library(difference_constraints)).

path(N) :-
        X #>= 0,
        X #=< N,
        reach(1,_,X,N).

reach(A,A,_X,_N).
reach(A,C,X,N) :-
        edge(A,B),
        NX #= X + 1,
        NX #=< N,
        reach(B,C,NX,N).

edge(30,16).
edge(30,14).
edge(30,27).
edge(30,22).
edge(30,29).
edge(30,6).
edge(30,5).
edge(30,23).
edge(30,19).
edge(30,7).
...
% the graph has 30 nodes and 742 edges
...
edge(1,5).
edge(1,27).
edge(1,29).
```

Figure A.2: Code of the path/1 program.

A.3 Fibonacci Function

fibo(N,F) is the well-known doubly recursive Fibonacci function. The code for CLP(Q) and difference constraints are slightly different, since difference constraints can have at most two variables per constraint.

```
%% - Call examples:
%%        fibo(10,F)
%%        fibo(N,89).

:- module(fibo,
        [
            fibo/2
        ], []).

:- use_package(library(tabling)).
:- const_table_module(library(
    difference_constraints/
    difference_constraints_tab)).
:- table fibo/2.

:- use_package(library(
    difference_constraints)).

fibo(N, F) :-
        N #= 0,
        F #= 0.
fibo(N, F) :-
        N #= 1,
        F #= 1.
fibo(N, F):-
        N #>= 2,
        N1 #= N - 1,
        N2 #= N - 2,
        F1 #=< F,
        F2 #=< F1,
        fibo(N1, F1),
        fibo(N2, F2),
        F #= F1 + F2.
```

Figure A.3: Code of the fibo/2 program using difference constraints.

```
%% - Call examples:
%%        fibo(10,F)
%%        fibo(N,89).

:- module(fibo,
        [
            fibo/2
        ], []).

:- use_package(library(tabling)).
:- const_table_module(library(clpq/
    clpq_tab)).
:- table fibo/2.

:- use_package(library(clpq)).

fibo(0, 0).
fibo(1, 1).
fibo(N, F):-
        N #> 2,
        N1 #= N - 1,
        N2 #= N - 2,
        F1 #> 0,
        F2 #> 0,
        F #= F1 + F2,
        fibo(N1, F1),
        fibo(N2, F2).
```

Figure A.4: Code of the fibo/2 program using CLP(Q).

A.4 Distance_bounded graph traversal (ep_e/3, pe_e/3, e_ep/3 and e_pe/3)

The four versions of dist(Origin, Dest, Length) compute paths from Origin to Dest. They are called with a bound on Length (Length # ≤ 200 in this case), on two graph: a graph with 10 nodes and 244 edges, and another cycled graph with 10 nodes and 245 edges.

```
%% USER MANUAL
%% - D .<. 200, ep_p(CX,Y,D) compute the
     distance between all the nodes
%%   which distance is less than 200
     units/

:- module(distance_bounded,
            [
                ep_e/3,
                pe_e/3,
                e_ep/3,
                e_pe/3
            ]).


:- use_package(library(tabling)).
:- const_table_module(library(clpq/
    clpq_tab)).
:- table ep_e/3, pe_e/3, e_ep/3, e_pe/3.

:- use_package(library(clpq)).

ep_e(A,B,D) :-
        D1 .>. 0,
        D2 .>. 0,
        D .=. D1 + D2,
        edge(A,Z,D1),
        ep_e(Z,B,D2).
ep_e(A,B,D) :-
        edge(A,B,D).

pe_e(A,B,D) :-
        D1 .>. 0,
        D2 .>. 0,
        D .=. D1 + D2,
        pe_e(Z,B,D2),
        edge(A,Z,D1).
pe_e(A,B,D) :-
        edge(A,B,D).
```

```
e_ep(A,B,D) :-
        edge(A,B,D).
e_ep(A,B,D) :-
        D1 .>. 0,
        D2 .>. 0,
        D .=. D1 + D2,
        edge(A,Z,D1),
        e_ep(Z,B,D2).

e_pe(A,B,D) :-
        edge(A,B,D).
e_pe(A,B,D) :-
        D1 .>. 0,
        D2 .>. 0,
        D .=. D1 + D2,
        e_pe(Z,B,D2),
        edge(A,Z,D1).

%% this edge create a cycle graph
edge(3, 1, 10).
%% this edge create a cycle graph

edge(10, 14, 240).
edge(10, 9, 190).
edge(10, 15, 250).
edge(10, 29, 390).
edge(10, 27, 370).
edge(10, 18, 280).
edge(10, 3, 130).
edge(10, 26, 360).
edge(10, 19, 290).
edge(10, 23, 330).
...
%  the graph has 10 nodes and 244 edges
...
edge(1, 5, 60).
edge(1, 27, 280).
edge(1, 29, 300).
```

Figure A.5: Code of the four version of the distance_bounded program.

A.5 Abstract Interpreter and Constraint over (Finite) Lattices

A.5.1 Abstract Interpreter (absint.pl)

This abstract interpreter is an simple analyzer base on abstract interpretation which uses tabling to automatically detect the fixpoint and terminate. The module absint.pl implements the fix-point algorithm using tabling. The predicate analyze/2 is the entry point to analyze the programs. The tabled predicate step_goal_att/1 guaranties that the interpreter reaches a fix-point.

```
%% USER MANUAL
%% - analyze(Program, Out): analyze the program Program and returns in
%%   Out a list with the abstract values of the arguments in Program
%%   Program should be define in stored_programs.pl
%% - call examples:
%%   analyze(permute(A,B,C,D), Out).
%% - program example:
%%   stored_program(permute(A1,A2,A3,A4), [A1 < 3, permute(A4,A1,A2,A3)]).
%%   ...

:- module(absint,
        [
            analyze/1
        ]).

:- use_package(library(tabling)).
:- const_table_module(library(lattice_solver/lattice_solver_tab)).
:- table step_goal_att/1.

:- use_package(library(lattice_solver)).

:- include('stored_programs').

analyze(Goal, Out) :-
        Goal =.. [_ | List],
        initialize(List),
        (
            step_att(Goal) ->
            true
        ;
            make_bot(List)
        ),
        get_abs_list(List, Out),
        abolish_all_tables.

initialize([]).
initialize([X | Xs]) :-
        abs(X, top),
        initialize(Xs).

interp_body_att([]).
interp_body_att([SubGoal | RestBody]) :-
        step_att(SubGoal),
        interp_body_att(RestBody).

step_goal_att(Goal) :-
        stored_program(Goal, Body),
        interp_body_att(Body).
```

```
step_att(Goal) :-
        Goal =.. [_ | List],
        findall(List,
        (
            step_goal_att(Goal)
        ), GoalClauses),
        join_info_att(GoalClauses, List).

step_att(V is E) :-     V abs_is E.

step_att(atom(V)) :-    abs_atom(V).

step_att(V1 = V2) :-    V1 .=. V2.

step_att(V1 < V2):-     V1 .<. V2.
step_att(V1 > V2):-     V1 .>. V2.
step_att(V1 =:= V2):-   V1 .=:=. V2.

step_att(V1 @< V2):-    V1 .@<. V2.
step_att(V1 @> V2):-    V1 .@>. V2.
step_att(V1 == V2):-    V1 .==. V2.

step_att(V1 .. AbsV1) :-V1 .<-. AbsV1.
```

Figure A.6: Code of the abstract interpreter (absint.pl).

A.5.2 Programs to be Analyzed (stored_program.pl)

This file stores the program to be analyze using the syntax: stored_program(head(A1, ... , An), [goal_1, ... , goal_n]).

```
%% USER MANUAL
%% - program example:
%%   stored_program(head(Arg1, ..., ArgN), [goal1, ... goalN]).
%%   ...

stored_program(permute(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10),
        [A1 < 3, permute(A10,A1,A2,A3,A4,A5,A6,A7,A8,A9)]).
stored_program(permute(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10),
        [A1 < 0, permute(A10,A1,A2,A3,A4,A5,A6,A7,A8,A9)]).
stored_program(permute(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10),
        [A1 = -3, A2 = -3, A3 = -3, A4 = -3, A5 = -3, A6 = -3, A7 = -3, A8 = -3, A9 = -3, A10
            = -3]).

stored_program(permute(A1,A2,A3,A4,A5,A6,A7,A8),
        [A1 < 3, permute(A8,A1,A2,A3,A4,A5,A6,A7)]).
stored_program(permute(A1,A2,A3,A4,A5,A6,A7,A8),
        [A1 < 0, permute(A8,A1,A2,A3,A4,A5,A6,A7)]).
stored_program(permute(A1,A2,A3,A4,A5,A6,A7,A8),
        [A1 = -3, A2 = -3, A3 = -3, A4 = -3, A5 = -3, A6 = -3, A7 = -3, A8 = -3]).

stored_program(permute(A1,A2,A3,A4,A5,A6), [A1 < 3, permute(A6,A1,A2,A3,A4,A5)]).
stored_program(permute(A1,A2,A3,A4,A5,A6), [A1 < 0, permute(A6,A1,A2,A3,A4,A5)]).
stored_program(permute(A1,A2,A3,A4,A5,A6), [A1 = -3, A2 = -3, A3 = -3, A4 = -3, A5 = -3, A6 =
    -3]).

stored_program(permute(A1,A2,A3,A4), [A1 < 3, permute(A4,A1,A2,A3)]).
stored_program(permute(A1,A2,A3,A4), [A1 < 0, permute(A4,A1,A2,A3)]).
stored_program(permute(A1,A2,A3,A4), [A1 = -3, A2 = -3, A3 = -3, A4 = -3]).


stored_program(permute_p(X1,X2,X3,X4), [X1 > 0, permute_q(X1,X2,X3,X4)]).
stored_program(permute_p(X1,X2,X3,X4), [X1 = 0, permute_q(X1,X2,X3,X4)]).
stored_program(permute_p(X1,X2,X3,X4), [X1 < 0, permute_q(X1,X2,X3,X4)]).
stored_program(permute_q(X1,X2,X3,X4), [permute_p(X2,X3,X4,X1)]).
stored_program(permute_q(X1,X2,X3,X4), [X1 < 0, permute_q(X2,X2,X3,X4)]).
stored_program(permute_q(X1,X2,X3,X4), [X1 < X2, X2 < X3, X2 < X4]).
```

Figure A.7: Code of the stored programs to analyze (stored_program.pl).

A.5.3 Constraints over (Finite) Lattices Solver (lattice_solver.pl)

The constraint solver can be use with two variants: with the lattice and domain operations written either in a *functional* style (e.g., inputs to abstract domain operations produce outputs) or as a constraint system, which makes it possible to issue richer queries. This file implements the lattice operations, the bridge with the abstract domain operation and the operation required by the interface Mod TCLP.

```
:- module(lattice_solver,
            [
                'constraint'/2,
                'abs'/2,
                'abs_atom'/1,
                'abs_is'/2,
                '.=.'/2,
                '.<.'/2,
                '.>.'/2,
                '.=:=.'/2,
                '.@<.'/2,
                '.@>.'/2,
                '.==.'/2,
                'join'/3,
                'meet'/3,
                'in'/2,

                get_abs_list/2,
                make_bot/1,
                join_info_att/2,

                aux_project_domain/3,
                aux_entail/4,
                aux_apply_answer/3,
                aux_clean_vars/2
            ]).

:- use_package(attr).
:- use_module(engine(attributes)).

:- use_module(library(terms_vars)).
:- use_module(library(sets)).

:- use_module(abstraction).
:- include(lattice_solver_syntax).

%% PROPAGATION ALGORITHM %%
add_constraint(V1, _, _AbsX) :-
        ground(V1), !.
add_constraint(V1, Constraints, NewAbs) :-
        (
            get_attr_local(V1, '$abs'(PrevConst, Abs)) ->
            insert_constraint(V1, Constraints, PrevConst, FinalCosnt),
            abs_meet(NewAbs, Abs, FinalAbs),
            FinalAbs \= bot,
            put_attr_local(V1, '$abs'(FinalCosnt, FinalAbs)),
            (
                Abs == FinalAbs ->
                true
            ;
```

```
                %               print('|'),
                check_constraint(PrevConst)
            )
        ;
            in_abs_domain(NewAbs, _),
            insert_constraint(V1, Constraints, [], FinalConst),
            put_attr_local(V1, '$abs'(FinalConst, NewAbs))
        ).

insert_constraint(V1, Constraint, Rs, Rs) :-
        varset(Constraint-V1, Xs),
        Xs == [V1], !.
insert_constraint(_, Constraint, Rs, Final) :-
        insert(Rs, Constraint, Final).
check_constraint([]).
check_constraint([C|Cs]) :-
        call(C),
        check_constraint(Cs).


%% OPERATORS %%
constraint(Constant, []) :-
        ground(Constant), !.
constraint(V1, Const) :-
        (
            get_attr_local(V1, '$abs'(Const, _)) ->
            true
        ;
            Const = []
        ).

abs(Constant, AbsX) :-
        ground(Constant), !,
        (
            abstract_constant(Constant, AbsX) ->
            true
        ;
            fail
        ).
abs(X, NewAbsX) :-
        ground(NewAbsX), !,
        (
            get_attr_local(X, '$abs'(PrevConst, Abs)) ->
            abs_meet(NewAbsX, Abs, FinalAbs),
            FinalAbs \= bot,
            put_attr_local(X, '$abs'(PrevConst, FinalAbs)),
            (
                Abs == FinalAbs ->
                true
            ;
                %               true
                check_constraint(PrevConst)
            )
        ;
            in_abs_domain(NewAbsX, _),
            put_attr_local(X, '$abs'([], NewAbsX))
        ).
abs(X, AbsX) :-
        (
            get_attr_local(X, '$abs'(_, PrevAbs)) ->
            AbsX = PrevAbs
        ;
```

```
            AbsX = top,
            put_attr_local(X, '$abs'([], AbsX))
        ).

abs_atom(V1) :-
        abs(V1, atm).

V1 'abs_is' E :-
        nonvar(E),
        E =.. [Op, S1, S2],
        abs(V1, AbsV1),
        abs(S1, AbsS1),
        abs(S2, AbsS2),
        abs_operation(Op, AbsV1, AbsS1, AbsS2, [NewAbsV1, NewAbsS1, NewAbsS2]),
        add_constraint(V1, V1 'abs_is' E, NewAbsV1),
        add_constraint(S1, V1 'abs_is' E, NewAbsS1),
        add_constraint(S2, V1 'abs_is' E, NewAbsS2).

V1 .=. V2 :-
        abs(V1, AbsV1),
        abs(V2, AbsV2),
        abs_asigment(=, AbsV1, AbsV2, [NewAbsV1, NewAbsV2]),
        add_constraint(V1, V1 .=. V2, NewAbsV1),
        add_constraint(V2, V1 .=. V2, NewAbsV2).

V1 .<. V2 :-
        abs(V1, AbsV1),
        abs(V2, AbsV2),
        abs_condition(<, AbsV1, AbsV2, [NewAbsV1, NewAbsV2]),
        add_constraint(V1, V1 .<. V2, NewAbsV1),
        add_constraint(V2, V1 .<. V2, NewAbsV2).
V1 .>. V2 :-
        V2 .<. V1.

V1 .=:=. V2 :-
        abs(V1, AbsV1),
        abs(V2, AbsV2),
        abs_condition(=:=, AbsV1, AbsV2, [NewAbsV1, NewAbsV2]),
        add_constraint(V1, V1 .=:=. V2, NewAbsV1),
        add_constraint(V2, V1 .=:=. V2, NewAbsV2).

%% var @< num @< atm @< str %%
V1 .@<. V2 :-
        abs(V1, AbsV1),
        abs(V2, AbsV2),
        abs_condition(@<, AbsV1, AbsV2, [NewAbsV1, NewAbsV2]),
        add_constraint(V1, V1 .@<. V2, NewAbsV1),
        add_constraint(V2, V1 .@<. V2, NewAbsV2).
V1 .@>. V2 :-
        V2 .@<. V1.

V1 .==. V2 :-
        abs(V1, AbsV1),
        abs(V2, AbsV2),
        abs_condition(==, AbsV1, AbsV2, [NewAbsV1, NewAbsV2]),
        add_constraint(V1, V1 .==. V2, NewAbsV1),
        add_constraint(V2, V1 .==. V2, NewAbsV2).


%% lattice constraints -
V1 in V2 :-
        abs(V1, AbsV1),
```

```
        abs(V2, AbsV2),
        abs_meet(AbsV1, AbsV2, NewAbsV1),
        add_constraint(V1, V1 in V2, NewAbsV1),
        add_constraint(V2, V1 in V2, AbsV2).

%% join /\ lub
join(A, B, S) :-
        abs(A, AbsA),
        abs(B, AbsB),
        abs_join(AbsA, AbsB, AbsS),
        AbsS \= bot,
        abs(S, AbsS).

%% meet \/ glb
meet(A, B, S) :-
        abs(A, AbsA),
        abs(B, AbsB),
        abs_meet(AbsA, AbsB, AbsS),
        AbsS \= bot,
        abs(S, AbsS).

%% Extras %%
join_abs_list([X],       X).
join_abs_list([X, Y|Rs], List) :-
        join_abs_list_aux(X, Y, Join),
        join_abs_list([Join|Rs], List).
join_abs_list_aux([],      [],      []).
join_abs_list_aux([X1|Xs], [Y1|Ys], [J|Js]) :-
        abs_join(X1, Y1, J),
        join_abs_list_aux(Xs, Ys, Js).
%% Extras %%



%% attr predicates %%
:- multifile attr_portray_hook/2.
:- multifile attr_unify_hook/2. %% needed
:- multifile attribute_goals/3.

attr_portray_hook(Att, Var) :-
        display(Var), display(', '), display(Att).

attr_unify_hook('$abs'(ConstX, AbsX), Y) :-
        get_attr_local(Y, '$abs'(ConstY, AbsY)),
        abs_meet(AbsX, AbsY, FinalAbs),
        ord_union(ConstX, ConstY, FinalConst),
        put_attr_local(Y, '$abs'(FinalConst, FinalAbs)),
        % check_constraint(FinalConst).
        (
            AbsX == AbsY ->
            true
        ;
            AbsX = FinalAbs ->
            check_constraint(ConstY)
        ;
            AbsY = FinalAbs ->
            check_constraint(ConstX)
        ;
            check_constraint(FinalConst)
        ).

attribute_goals(X, [S|T], T) :-
```

```
        get_attr_local(X, '$abs'(Consts, Dom)),
        S = abs([X, Consts, Dom]).
%% attr pedricates %%




%% Auxiliar predicates used by the analyzer %%
%% Join the abstract values from all the clauses of a Goal %%
join_info_att([A], Copy) :-
        copy_list(A, Copy).
join_info_att([A, B|Rest], Rs) :-
        join_info_list(A, B, Join),
        join_info_att([Join|Rest], Rs).
join_info_list([],     [],     []).
join_info_list([A|As], [B|Bs], [Join|Js]) :-
        join(A, B, Join),
        join_info_list(As, Bs, Js).
copy_list([],       []).
copy_list([A|Rest], [Copy|RestCopy]) :-
        abs(A,    AbsA),
        abs(Copy, AbsA),
        copy_list(Rest, RestCopy).

%% To get the solution of the analyzer %%
get_abs_list([],     []).
get_abs_list([V|Vs], [abs(V, AbsV)|AbsList]) :-
        (
            get_attr_local(V, '$abs'(_, AbsV)) ->
            detach_attribute(V)
        ;
            AbsV = var
        ),
        get_abs_list(Vs, AbsList).

%% To give the answer in case the program has no success output %%
make_bot([]).
make_bot([V|Rest]) :-
        put_attr_local(V, '$abs'([], bot)),
        make_bot(Rest).
%% Auxiliar predicates used by the analyzer %%




%% Aux predicate for tabling interface:
%% PROJECT DOMAIN (AND CONSTRAINTS) %%
aux_project_domain(Vars, Space, Dom) :-
        get_domain(Vars, Dom),
        dump_constraint(Vars, Space).

get_domain([], []).
get_domain([X|Xs], [D|Ds]) :-
        abs(X, D),
        get_domain(Xs, Ds).

dump_constraint(Vars, [Vars,Space]) :-
        sort(Vars, V),
        project(V, Others, Space1),
        simplifier(Space1, Others, Space).
```

```
project(Vars, TotalOthers, Space) :-
        take_in_const(Vars, C0),
        others(Vars, C0, Others),
        project_aux(Vars, C0, Others, [], TotalOthers, Space).

project_aux(_,    C,  [],      O,         O,           C) :- !.
project_aux(Vars, C0, Others0, PreOthers, TotalOthers, Space) :-
        take_in_const(Others0, C1),
        ord_union(C0,   C1,        C2),
        ord_union(Vars, PreOthers, Total0),
        others(Total0, C2, Others1),
        ord_union(PreOthers, Others1, PreOthers1),
        project_aux(Vars, C2, Others1, PreOthers1, TotalOthers, Space).

take_in_const([],     []).
take_in_const([X|Xs], Const) :-
        constraint(X, Cs),
        filter_in(Cs, In_Cs),
        take_in_const(Xs, CXs),
        ord_union(CXs, In_Cs, Const).

filter_in([],     []).
filter_in([X|Xs], [X|Fs]) :-
        nonvar(X),
        function(X, in, 2),
        filter_in(Xs, Fs).
filter_in([_X|Xs], Fs) :-
        filter_in(Xs, Fs).

others(Vars, InConst, Other) :-
        varset(InConst, Total),
        ord_subtract(Total, Vars, Other).

simplifier(Space1, [],     Space1).
simplifier(Space1, [O|Os], FinalSpace) :-
        split_store(Space1, O, SpaceLeft, SpaceRight, NewSpace),
        new_constraint(SpaceLeft, SpaceRight, NewConstraints),
        ord_union(NewSpace, NewConstraints, NextSpace),
        simplifier(NextSpace, Os, FinalSpace).

split_store([],             _O, [],             [], []).
split_store([in(O1, V)|Ss], O,  [in(O1, V)|Ls], Rs, Ns) :-
        O1 == O, !,
        split_store(Ss, O, Ls, Rs, Ns).
split_store([in(V, O1)|Ss], O, Ls, [in(V, O1)|Rs], Ns) :-
        O1 == O, !,
        split_store(Ss, O, Ls, Rs, Ns).
split_store([C|Ss], O, Ls, Rs, [C|Ns]) :-
        split_store(Ss, O, Ls, Rs, Ns).

new_constraint([],            _,  []).
new_constraint([in(_, V)|Ls], Rs, NewC) :-
        new_constraint_aux(V, Rs, NewC0),
        new_constraint(Ls, Rs, NewC1),
        ord_union(NewC0, NewC1, NewC).

new_constraint_aux(_, [],             []).
new_constraint_aux(V, [in(V2, _)|Rs], [in(V2, V)|Ss]) :-
        V \== V2, !,
        new_constraint_aux(V, Rs, Ss).
new_constraint_aux(V, [_|Rs], Ss) :-
```

```
        new_constraint_aux(V, Rs, Ss).

%% ENTAIL %%
aux_entail(SpaceA, DomainA, SpaceB, DomainB) :-
        entail_domain(DomainA, DomainB),
        entail_store(SpaceA, SpaceB).

entail_domain([], []).
entail_domain([X|Xs], [Y|Ys]) :-
        abs_meet(X, Y, X),
        entail_domain(Xs, Ys).

entail_store([Vars,SpaceA], [Vars,SpaceB]) :-
        sort(SpaceA, SortA),
        sort(SpaceB, SortB),
        ord_subtract(SortA, SortB, []).

%% APPLY ANSWER %%
aux_apply_answer(Vars, Space, Dom) :-
        apply_domain(Vars, Dom),
        apply_store(Vars, Space).

apply_domain([], []).
apply_domain([X|Xs], [D|Ds]) :-
        check_answer(X, D),
        apply_domain(Xs, Ds).
apply_store(Vars, [Vars, Constraints]) :-
        call_constraints(Constraints).
call_constraints([]).
call_constraints([X |Xs]) :-
        call(X),
        call_constraints(Xs).
check_answer(X, NewAbsX) :-
        var(X),
        get_attribute(X, att(X, _, [])), !,
        put_attr_local(X, '$abs'([], NewAbsX)).
check_answer(X, NewAbsX) :-
        get_attr_local(X, '$abs'(Const, AbsX)), !,
        abs_meet(NewAbsX, AbsX, FinalAbsX),
        put_attr_local(X, '$abs'(Const, FinalAbsX)).
check_answer(X, NewAbsX) :-
        put_attr_local(X, '$abs'([], NewAbsX)).


%% CLEAN THE CONSTRAINTS FROM THE ABSTRACT DOMAIN %%
aux_clean_vars(Vars, [Vars,Space]) :-
        clean_constraints(Vars),
        call_constraints(Space).
clean_constraints([]).
clean_constraints([X|Xs]) :-
        get_attr_local(X, '$abs'(_, AbsX)),
        put_attr_local(X, '$abs'([], AbsX)),
        clean_constraints(Xs).
```

Figure A.8: Code of the constraint over (finite) lattices solver (lattice_solver.pl).

A.5.4 The Sign Abstract Domain (abstraction.pl)

This module implements the sign abstract domain operations. The use of truth_table/1 for declarativeness reduces the performance, but the use of tabling to memorize the calls to calculate/4 alleviates this overhead.

```
:- module(abstraction, [
                in_abs_domain/2,
                abstract_constant/2,
                abs_operation/5,
                abs_asigment/4,
                abs_condition/4,
                abs_join/3,
                abs_meet/3,
                calculate/4,
                abs_join_list/2
            ]).


% :- use_package(library(tabling)).
% :- table calculate/4.


%% Abstractions
in_abs_domain(X, _) :- member(X, [top, bot]).
in_abs_domain(X, numbers) :- member(X, [neg, pos, zero, neg+z, pos+z, not_z,
                num]).
in_abs_domain(var, variables).
in_abs_domain(atm, atoms).
in_abs_domain(str, structures).

%% Abstraction of constants
abstract_constant(0, zero).
abstract_constant(N, pos) :- num(N), N > 0.
abstract_constant(N, neg) :- num(N), N < 0.
abstract_constant(A, atm) :- atom(A).
abstract_constant(S, str) :- struct(S).

%% Abstract operations. Use truth tables.
abs_operation(Op, AbsV1, AbsS1, AbsS2, NewAbstractions) :-
        abs_meet(AbsS1, num, NumS1),
        abs_meet(AbsS2, num, NumS2),
        calculate(Op, NumS1, NumS2, NewAbsV1),
        (
            AbsV1 == var ->
            NewAbstractions = [NewAbsV1, NumS1, NumS2]
        ;
            abs_meet(AbsV1, NewAbsV1, NewAbsV1) ->
            NewAbstractions = [NewAbsV1, NumS1, NumS2]
        ;
            abs_meet(AbsV1, NewAbsV1, AbsV1) ->
            findall([TempS1, TempS2],
            (
                calculate(Op, TempS1, TempS2, AbsV1),
                abs_meet(TempS1, NumS1, TempS1),
                abs_meet(TempS2, NumS2, TempS2)
            ),
            All),
            (
                abs_join_list(All, [NewAbsS1, NewAbsS2]) ->
```

```
                NewAbstractions = [AbsV1, NewAbsS1, NewAbsS2]
            ;
                NewAbstractions = [AbsV1, NumS1, NumS2]
            )
        ;
            NewAbstractions = [bot, bot, bot]
        ).


abs_asigment(=, var,   AbsV2, [AbsV2, AbsV2]).
abs_asigment(=, AbsV1, var,   [AbsV1, AbsV1]) :-
        AbsV1 \= var.
abs_asigment(=, AbsV1, AbsV2, [NewAbsV1, NewAbsV1]) :-
        AbsV1 \= var, AbsV2 \= var,
        calculate(==, AbsV1, AbsV2, NewAbsV1).

abs_condition(=:=, AbsV1, AbsV2, [NewAbsV1, NewAbsV1]) :-
        abs_meet(AbsV1, num, NumV1),
        abs_meet(AbsV2, num, NumV2),
        calculate(==, NumV1, NumV2, NewAbsV1).
abs_condition(<, AbsV1, AbsV2, [NewAbsV1, NewAbsV2]) :-
        abs_meet(AbsV1, num, NumV1),
        abs_meet(AbsV2, num, NumV2),
        calculate(@<, NumV1, NumV2, NewAbsV1),
        calculate(@>, NumV2, NumV1, NewAbsV2).

abs_condition(==, AbsV1, AbsV2, [NewAbsV1, NewAbsV1]) :-
        calculate(==, AbsV1, AbsV2, NewAbsV1).
abs_condition(@<, AbsV1, AbsV2, [NewAbsV1, NewAbsV2]) :-
        calculate(@<, AbsV1, AbsV2, NewAbsV1),
        calculate(@>, AbsV2, AbsV1, NewAbsV2).

% join or lub - least upper bound
abs_join(AbsV1, AbsV2, Res) :-
        calculate(lub, AbsV1, AbsV2, Res).

% meet ot glb - greatest lower bound
abs_meet(Abs1, Abs2, Res) :-
        calculate(glb, Abs1, Abs2, Res).

%% Predicate for all the operators (also lub and glb)
calculate(Operator, AbsA, AbsB, AbsOut) :-
        truth_table([Operator, L1|Table]),
        member([AbsB|L2], Table),
        member(AbsA,      L1),
        calculate_aux(AbsA, L1, L2, Lout),
        member(AbsOut, Lout).
calculate_aux(_,     [],        [],     []).
calculate_aux(AbsA, [AbsA|L1], [X|L2], [X|LX]) :-
        calculate_aux(AbsA, L1, L2, LX).
calculate_aux(AbsA, [B|L1], [_L|L2], X) :-
        AbsA \= B,
        calculate_aux(AbsA, L1, L2, X).

abs_join_list([X], X).
abs_join_list([A, B| Ls], JoinList) :-
        abs_join_list_aux(A, B, Join),
        abs_join_list([Join| Ls], JoinList).
abs_join_list_aux([],[],[]).
abs_join_list_aux([A|As], [B|Bs], [J|Js]) :-
        abs_join(A, B, J),
        abs_join_list_aux(As, Bs, Js).
```

```
%% True Tables %%
truth_table([+,
       [bot, neg, zero,  pos, neg+z, not_z, pos+z, num, var, atm, str, top],
[bot,   bot, bot, bot,   bot, bot,   bot,   bot,   bot, bot, bot, bot, bot],
[neg,   bot, neg, neg,   num, neg,   num,   num,   num, bot, bot, bot, num],
[zero,  bot, neg, zero,  pos, neg+z, not_z, pos+z, num, bot, bot, bot, num],
[pos,   bot, num, pos,   pos, num,   num,   pos,   num, bot, bot, bot, num],
[neg+z, bot, neg, neg+z, num, neg+z, num,   num,   num, bot, bot, bot, num],
[not_z, bot, num, not_z, num, num,   not_z, num,   num, bot, bot, bot, num],
[pos+z, bot, num, pos+z, pos, num,   num,   pos+z, num, bot, bot, bot, num],
[num,   bot, num, num,   num, num,   num,   num,   num, bot, bot, bot, num],
[var,   bot, bot, bot,   bot, bot,   bot,   bot,   bot, bot, bot, bot, bot],
[atm,   bot, bot, bot,   bot, bot,   bot,   bot,   bot, bot, bot, bot, bot],
[str,   bot, bot, bot,   bot, bot,   bot,   bot,   bot, bot, bot, bot, bot],
[top,   bot, num, num,   num, num,   num,   num,   num, bot, bot, bot, num]]).

truth_table([*,
       [bot, neg,   zero, pos,   neg+z, not_z, pos+z, num,  var,atm, str, top],
[bot,   bot, bot,   bot,  bot,   bot,   bot,   bot,   bot,  bot,bot, bot, bot],
[neg,   bot, pos,   zero, neg,   pos+z, not_z, neg+z, num,  bot,bot, bot, num],
[zero,  bot, zero,  zero, zero,  zero,  zero,  zero,  zero, bot,bot, bot, num],
[pos,   bot, neg,   zero, pos,   neg+z, not_z, pos+z, num,  bot,bot, bot, num],
[neg+z, bot, pos+z, zero, neg+z, pos+z, num,   neg+z, num,  bot,bot, bot, num],
[not_z, bot, not_z, zero, not_z, num,   not_z, num,   num,  bot,bot, bot, num],
[pos+z, bot, neg+z, zero, pos+z, neg+z, num,   pos+z, num,  bot,bot, bot, num],
[num,   bot, num,   zero, num,   num,   num,   num,   num,  bot,bot, bot, num],
[var,   bot, bot,   bot,  bot,   bot,   bot,   bot,   bot,  bot,bot, bot, bot],
[atm,   bot, bot,   bot,  bot,   bot,   bot,   bot,   bot,  bot,bot, bot, bot],
[str,   bot, bot,   bot,  bot,   bot,   bot,   bot,   bot,  bot,bot, bot, bot],
[top,   bot, num,   num,  num,   num,   num,   num,   num,  bot,bot, bot, num]]).

% == ,[As...],
% B  , X1
% ...
% means A == B -> newSubsA = X1. Also newSubsB = X1
truth_table([==,
       [bot, neg, zero, pos, neg+z, not_z, pos+z, num,   var, atm, str, top],
[bot,   bot, bot, bot,  bot, bot,   bot,   bot,   bot,   bot, bot, bot, bot],
[neg,   bot, neg, bot,  bot, neg,   neg,   bot,   neg,   bot, bot, bot, neg],
[zero,  bot, bot, zero, bot, zero,  bot,   zero,  zero,  bot, bot, bot, zero],
[pos,   bot, bot, bot,  pos, bot,   pos,   pos,   pos,   bot, bot, bot, pos],
[neg+z, bot, neg, zero, bot, neg+z, neg,   zero,  neg+z, bot, bot, bot, neg+z],
[not_z, bot, neg, bot,  pos, neg,   not_z, pos,   not_z, bot, bot, bot, not_z],
[pos+z, bot, bot, zero, pos, zero,  pos,   pos+z, pos+z, bot, bot, bot, pos+z],
[num,   bot, neg, zero, pos, neg+z, not_z, pos+z, num,   bot, bot, bot, num],
[var,   bot, bot, bot,  bot, bot,   bot,   bot,   bot,   var, bot, bot, var],
[atm,   bot, bot, bot,  bot, bot,   bot,   bot,   bot,   bot, atm, bot, atm],
[str,   bot, bot, bot,  bot, bot,   bot,   bot,   bot,   bot, bot, str, str],
[top,   bot, neg, zero, pos, neg+z, not_z, pos+z, num,   var, atm, str, top]]).

% @< ,[As...],
% B , X1
% ...
% means A < B -> newSubsA = X1. To know newSubsB check B > A
truth_table([@<,
       [bot, neg, zero, pos, neg+z, not_z, pos+z, num, var, atm, str, top],
[bot,   bot, bot, bot,  bot, bot,   bot,   bot,   bot, bot, bot, bot, bot],
[neg,   bot, neg, bot,  bot, neg,   neg,   bot,   neg, var, bot, bot, top],
[zero,  bot, neg, bot,  bot, neg,   neg,   bot,   neg, var, bot, bot, top],
[pos,   bot, neg, zero, pos, neg+z, not_z, pos+z, num, var, bot, bot, top],
[neg+z, bot, neg, bot,  bot, neg,   neg,   bot,   neg, var, bot, bot, top],
```

```
[not_z, bot, neg, zero, pos, neg+z, not_z, pos+z, num, var, bot, bot, top],
[pos+z, bot, neg, zero, pos, neg+z, not_z, pos+z, num, var, bot, bot, top],
[num,   bot, neg, zero, pos, neg+z, not_z, pos+z, num, var, bot, bot, top],
[var,   bot, bot, bot,  bot, bot,   bot,   bot,   bot, var, bot, bot, var],
[atm,   bot, neg, zero, pos, neg+z, not_z, pos+z, num, var, atm, bot, top],
[str,   bot, neg, zero, pos, neg+z, not_z, pos+z, num, var, atm, str, top],
[top,   bot, neg, zero, pos, neg+z, not_z, pos+z, num, var, atm, str, top]]).

% @>  ,[As...],
% B  , X1
% ...
% means A > B -> newSubsA = X1. To know newSubsB check B < A
truth_table([@>,
       [bot, neg, zero, pos, neg+z, not_z, pos+z, num, var, atm, str, top],
[bot,   bot, bot, bot,  bot, bot,   bot,   bot,   bot, bot, bot, bot, bot],
[neg,   bot, neg, zero, pos, neg+z, not_z, pos+z, num, bot, atm, str, top],
[zero,  bot, bot, bot,  pos, bot,   pos,   pos,   pos, bot, atm, str, top],
[pos,   bot, bot, bot,  pos, bot,   pos,   pos,   pos, bot, atm, str, top],
[neg+z, bot, neg, zero, pos, neg+z, not_z, pos+z, num, bot, atm, str, top],
[not_z, bot, neg, zero, pos, neg+z, not_z, pos+z, num, bot, atm, str, top],
[pos+z, bot, bot, bot,  pos, bot,   pos,   pos,   pos, bot, atm, str, top],
[num,   bot, neg, zero, pos, neg+z, not_z, pos+z, num, bot, atm, str, top],
[var,   bot, neg, zero, pos, neg+z, not_z, pos+z, num, var, atm, str, top],
[atm,   bot, bot, bot,  bot, bot,   bot,   bot,   bot, bot, atm, str, top],
[str,   bot, bot, bot,  bot, bot,   bot,   bot,   bot, bot, bot, str, str],
[top,   bot, neg, zero, pos, neg+z, not_z, pos+z, num, var, atm, str, top]]).

truth_table([lub,
      [bot,  neg,  zero, pos,  neg+z,not_z,pos+z,num,var,atm,str,top],
[bot,  bot,  neg,  zero, pos,  neg+z,not_z,pos+z,num,var,atm,str,top],
[neg,  neg,  neg,  neg+z,not_z,neg+z,not_z,num,  num,top,top,top,top],
[zero, zero, neg+z,zero, pos+z,neg+z,num,  pos+z,num,top,top,top,top],
[pos,  pos,  not_z,pos+z,pos,  num,  not_z,pos+z,num,top,top,top,top],
[neg+z,neg+z,neg+z,neg+z,num,  neg+z,num,  num,  num,top,top,top,top],
[not_z,not_z,not_z,num,  not_z,num,  not_z,num,  num,top,top,top,top],
[pos+z,pos+z,num,  pos+z,pos+z,num,  num,  pos+z,num,top,top,top,top],
[num,  num,  num,  num,  num,  num,  num,  num,  num,top,top,top,top],
[var,  var,  top,  top,  top,  top,  top,  top,  top,var,top,top,top],
[atm,  atm,  top,  top,  top,  top,  top,  top,  top,top,atm,top,top],
[str,  str,  top,  top,  top,  top,  top,  top,  top,top,top,str,top],
[top,  top,  top,  top,  top,  top,  top,  top,  top,top,top,top,top]]).

truth_table([glb|X]) :- truth_table([== |X]).
```

Figure A.9: Code of the abstract domain and its operations (abstraction.pl).

A.5.5 Constraints over (Finite) Lattices: Mod TCLP Interface (lattice_solver_tab.pl)

Since the predicates required by the projection, entailment and application operations are defined in the constraint solver, the interface is a bridge which execute the required predicates from the constraint solver.

```
:- module(lattice_solver_tab,
            [
                project_domain/2,
                entail/4,
                apply_answer/3,
                project_gen_store/3,
                project_answer_store/3,
                current_store/1,
                reinstall_store/3
            ]).

:- use_module(library(lattice_solver)).

%% tabling interface %%
project_domain(Vars,(Space,Dom)) :-
        aux_project_domain(Vars,Space,Dom).
entail(_V, (SpaceA,DomA),(SpaceB,DomB), _Space) :-
        \+ \+ aux_entail(SpaceA,DomA,SpaceB,DomB).
apply_answer(Vars,(SpaceA,DomA),_Space) :-
        aux_apply_answer(Vars,SpaceA,DomA).

project_gen_store(Vars, (SpaceA, _D), _P) :-
        aux_clean_vars(Vars, SpaceA).
project_answer_store(_X, _A, _P).

current_store(_P).
reinstall_store(_X, _O, _Orig).
%% tabling interface %%
```

Figure A.10: Code of the Mod TCLP interface (lattice_solver_tab.pl).